コンピュータ・サイエンス研究書シリーズ ㊵

アルゴリズム＋データ構造＝PASCAL プログラム

Niklaus Wirth 著
東京工業大学助教授・工博　片山卓也訳

日本コンピュータ協会

コンピュータ・サイエンス研究書シリーズ ㊵

アルゴリズム＋データ構造 ＝プログラム

Niklaus Wirth 著
東京工業大学助教授・工博 片山卓也訳

日本コンピュータ協会

ALGORITHMS + DATA STRUCTURES = PROGRAMS

NIKLAUS WIRTH

Eidgenossische Technische Hochschule
Zurich, Switzerland

はじめに

近年，**計算機プログラミング**は，その修得が多くの工学的プロジェクトの成功にとって基本的であり，またその科学的な取り扱いや表現が可能な課題であることが認められてきた．すなわち職人芸から専門的技術へと進化してきた．この進化に対する顕著な貢献は，まず E. W. Dijkstra, C. A. R. Hoare によってなされた．Dijkstra の “Notes on Structured Programming” は，プログラミングに科学的主題および知的挑戦としての新しい観点を与えたものであり，プログラミングにおけるひとつの革命に値するものである．Hoare の “Axiomatic Basis of Computer Programming” は，プログラムが数学的推論にもとづく厳密な解析が適用可能なものであることを平易に示している．これらの論文では，プログラマが直感的にかつ無意識に使ってきた手法やテニックを彼らに意識させることにより，プログラムにおける誤りの多くが防げることが，説得力のある論法で述べられている．また，これらの論文では，プログラムの合成と分解の局面に，より明確にいえば，プログラム・テキストで表現されるアルゴリズムの構造に，その注意が向けられている．ましてや複雑なデータ集合を扱う大規模で複雑なプログラムを構成する際には，系統的で科学的な手法が基本的に重要なことは極めて明らかである．それゆえ，プログラミングの方法論には，データの構造化に関するどのような局面も含まれていなければならない．**プログラム**とは，結局のところ，抽象的なアルゴリズムを特定な**データ**表現・構造の上に具体的に定式化したものである．データ構造はその用語や概念に非常に多くの多様性をもっているが，それらを整理した点で Hoare の論文 “Notes on Data Structuring” の貢献は際立っている．そこでは，データの構造化に関する決定は，そのデータに適用されるアルゴリズムが何であるかを知らずには不可能なこと，またその逆に，アルゴリズムの構造とその選択は，しばしばデータの構造に強く依存するものであることが述べられている．簡単にいえば，プログラムの構成とデータの構造とは不可分な関係にある．

それにもかかわらず，本書ではデータ構造に関する章から始めるが，それには2つの理由がある．第一には，アルゴリズムより先にデータが存在するとい

う直感を誰でも持っていることである．すなわち，あるものに演算をほどにす前にはそのものが存在しなくてはならない．第二には，そしてこれが直接の理由であるが，本書では読者に計算機のプログラミングに関する基本的概念を仮定していることである．伝統的に，そして意味のあることであるが，プログラミングの入門課程では，比較的簡単な構造のデータを操作するアルゴリズムに重点が置かれている．したがって，データ構造についての入門的な章から始めることは適当であろう．

本書を通じて，また特に第1章では，Hoare† によって論じられまたプログラム言語 PASCAL†† で実現されている理論と用語にしたがっている．この理論の本質は，最初の段階のデータは現実の現象を抽象化したものであり，必ずしも一般のプログラム言語では実現されない抽象的構造として定式化するのがよいということである．プログラムを構成してゆく過程で，このデータ表現は，プログラムが精密化されるのに合わせて除々に精密化され，利用可能なプログラム言語に課せられた制約条件を満たすように仕立てあげられていく．††† それゆえ，われわれはデータ構造の基本的構成原理をいくつか仮定し，これを**基本構造**とよぶ．ここで最も重要なことは，それらのものが実際の計算機上で容易に実現可能なものであるということである．なぜなら，もしそうでないと，それらはデータ記述の精密化の最終段階から生じる**分子**（molecule）としての実際のデータ表現要素とは考えられないからである．これらの基本構造とは，**レコード**，**配列**（固定長の）および**集合**である．これらの基本構成原理が，数学的概念としても基本的なものであることは驚くにはあたらない．

データ構造についてのこの理論の基礎は，基本構造と“高度な”構造を区別するところにある．前者は（それ自体原子要素から構成されるが）後者を構成している分子である．基本構造の変数は，その値を変えることはあっても，その構造やそれがとる値の集合を変えることは決してない．その結果として，その占める記憶領域の大きさは不変である．それにひきかえ，“高度な”構造は，プログラムの実行に際して，その値と構造とを変化させることで特徴づけられる．それゆえに，その実現のためにはより高度な手法が必要とされる．

順ファイル（または単に列）は，この分類では中間的位置にある．確かにそれは長さを変えるが，構造に対するその変化は自明なものである．順ファイルは，実際のどの計算機システムでも基本的役割をになうものであるので，第1

---

† “Notes on Data Structuring”

†† N. Wirth, “The Programming Language Pascal”, *Acta Informatica*, **1**, No. 1 (1971).

††† N. Wirth, “Programming Developement by Stepwise Refinement” *Comm. ACM*, **14**, No. 4 (1971) 221〜27.

章では基本構造のなかに含められている.

第2章では**ソーティング・アルゴリズム**を扱う.そこでは,同一の目的のための多くの異なった方法が示されている.これらのアルゴリズムの利害得失について数学的解析がなされているが,これは与えられた問題に対して良い解を選択することの重要性をプログラマに教えるためである.配列中のデータをソートする方法と,ファイルをソートする方法(これらは通常,内部ソートおよび外部ソートと呼ばれている)とに分けられているが,これはデータの表現法が,それに適用されるアルゴリズムやその複雑さに決定的な影響を与えていることを示している.ソーティングのために非常に多くの頁数が費されている理由は,これがプログラミングにおける非常に多くの原理や,他の多くの応用問題で生じる状況を示すのに理想的な題材であるからである.ソーティングについての例題だけを用いても,プログラミングの課程をひとつ作れるように思われることもしばしばあるくらいである.

入門的なプログラミングの課程で普通扱われないもうひとつの題目として再帰があるが,これはアルゴリズムの構成という立場からは重要な役割をもつものである.したがって第3章は,**再帰的アルゴリズム**にあてられている.

再帰は繰返しの一般化として示されているが,それゆえにプログラミングにおける重要で強力な概念である.不幸なことに,プログラミングに関する多くの教材では,単純な繰返しでも十分な例題を用いてその説明を行なっている.これに反し,第3章では再帰を使うことによって,解を最も自然に表現でき,繰返しでは不明確でわずらわしいプログラムになってしまうような例題だけを扱っている.**バックトラック・アルゴリズム**が再帰の理想的応用として述べられているが,再帰の最も自然な使い方は,その構造が再帰的に定義されたデータを操作するアルゴリズムに対してであろう.これらの例は,最後の章で扱われるが,第3章はそのための予備知識を与えるものである.

第4章では,**動的データ構造**すなわちプログラムの実行に伴ってその構造を変えるようなデータを扱う.再帰的データ構造は,通常使われる動的な構造の重要なサブクラスであることが示されている.再帰的定義はこの場合にも自然であり,また可能であるが,実際には使われないことが多い.その代わり,陽な参照あるいは**ポインタ**変数をプログラマに使用させることにより,その実現に使われる機構が明らかにされている.本書でもこの手法に従い,その技法の現状について述べる.第4章は,ポインタを用いたプログラミング,リスト構造,木構造,さらにはもっと複雑にからみ合ったデータに関する例題等にあてられており,通常“リスト処理”と呼ばれている(この呼び方は必ずしも適当

ではない）事柄が述べられている．木，特に探索木の構成法には相応の紙面がさかれている．最後に，探索木より好ましいことが多い分散表，あるいは“ハッシュ・コード”が述べられているが，これにより，応用上頻繁に現れる問題に対して，基本的に異なった2つの手法を比較することができる．

最後の章は，**形式言語**の定義や，**構文解析**の簡潔な導入，簡単な計算機用の小規模で簡単な言語の**コンパイラ**の構成から成っている．この章を設けた理由は3つある．その第1は，一人前のプログラマなら，プログラム言語のコンパイル過程に関する基本的問題や技法について，ある程度の知識を持っているべきであること．第2は，アプリケーションによっては，それをうまく実行するには，簡単な入力・制御言語を定義することが必要な場合が次第に増えていること．そして第3は，形式言語は記号列上の再帰構造を定めており，したがって，その処理系は再帰的手法の適用が有利であるような優れた実例になっていることである．再帰的手法は，プログラムが大きく，また巨大にさえなりがちな分野では，見通しのよい構造を得るために決定的に重要である．簡単な言語としては，PL/0 とよばれる言語が選ばれたが，この理由は，簡単すぎて有効な実例とはなり得ないような言語と，コンパイラの専門書にしか載せられないくらいそのコンパイラが大きくなってしまう言語とをバランスさせた結果である．

プログラミングは構成的な技術である．一体，構成的で創造的な活動はどのような方法で教えたらよいであろうか．ひとつの方法は，多数の実例から基本構成原理を抽出し，それを組織的に教えることである．しかしながらプログラミングは，複雑な知的活動と関連した非常に多様性のある分野である．したがって，純粋な“秘訣”のたぐいのみを教えればよいという考え方は間違っている．このとき残る方法は，特に優れた実例を注意深く選択し提示することである．全くあたりまえのことではあるが，実例の学習からすべての人が同量のものを学びとれると考えるべきではなく，学生の勤勉と直感力にゆだねられる部分の多いのがこの方法の特徴である．プログラムが比較的複雑で長い場合には，特にそうである．このようなプログラムが本書に含まれているのは偶然ではない．長いプログラム程実際問題ではあたりまえであり，さらに，明確には捕えにくいが本質的な要素である，(プログラムの) スタイルとか規則的な構造というものを示すには適している．それらには，またプログラムを読む技術の練習の意図も含まれているが，これはプログラムを書くことに比較してほとんど無視されている．これが長いプログラム全体を含ませた最初の動機である．読者にはプログラムが除々に作られてゆく過程が示され，プログラムが展開してい

く途中のスナップショットが与えられる．これによって，この展開が細部の**段階的詳細化** (stepwise refinement) であることが明らかとなろう．私は，細部に十分な注意を払いながらプログラムの最終的な形を示すことが必須であると考えるが，これはプログラミングにおいては，"悪魔"は細部に隠れているからである．もちろん学究的な人には，アルゴリズムの原理やその数学的解析だけでも興味はあるだろうが，これでは工学的な実務家には不親切であろう．それゆえ，私は実際に計算機上で実行できるような言語で最終プログラムを表現するという原則を厳密に守った．

もちろんこのときには，計算機で実行可能であると同時に，このような書物に載せられるような十分に機械独立な言語を見つける問題が生じる．この点については，現在広く使用されている言語，あるいは抽象的表記法はいずれも十分ではなかった．プログラム言語 PASCAL はその妥協案として適当なものであろう．PASCAL はこのような目的を考えて作られた言語であり，それゆえに本書を通して使用されている．本書を読み進んでゆけば，PASCAL の記法は簡単に理解できるので，ALGOL 60 や PL/I のような高級言語に精通しているプログラマなら，PASCAL プログラムを容易に理解することができる．しかし，多少の準備はそれなりの利益があろう．このためには，「Systematic Programming」[†] が適当であろう．なぜなら，これもまた PASCAL の記法に基づいているからである．しかし，この本は PASCAL 言語のマニアルとして書かれたものではなく，その目的のためには，さらに適当な本[††]がある．

本書は，チューリッヒ連邦工科大学 (ETH) のいくつかのプログラミング・コースで教えたものを縮約し，練り直したものである．本書中の多くのアイディアや見解は，ETH の同僚との討論から生まれたものである．特に，原稿に注意深く目を通してくれた Sandmayr 氏，原稿のタイプを注意深く忍耐強く行なってくれた Heidi Theiler 嬢に感謝する．また，IFIP 作業グループ 2.1, 2.3 の会合，特に，その折での E. W. Dijkstra や C. A. R. Hoare との数多くの活気ある議論も忘れ得ぬものであり，その影響にも言及したい．最後に，ETH が寛大にも提供してくれた環境や計算施設によるところも多く，これなしには本書の執筆は不可能であったろう．

N. WIRTH

† N. Wirth (Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1973).
†† K. Jensen and N. Wirth, "PASCAL—User Manual and Report" *Lecture Notes in Computer science*, Vol. 18 (Berlin, New York; Springer-Verlag, 1974).

訳　者　序　文

本書は，データ構造とプログラム作成技法に関する教科書であり，世界的にも極めて高い評価を受けているものである．

計算機科学教育におけるデータ構造の重要性は，いまさらここで述べることもないが，最も重要な基礎科目のひとつであることに異論はないであろう．したがって，これまでにもデータ構造に関する数多くの教科書が書かれており，それなりに優れた特徴をもっているものも少くないが，データ構造自体に関する説明や記述が主であり，それを用いていかにプログラミングを行なうかという視点に欠けているように見受けられる．

本来，プログラムの構造は，それが操作しようとしているデータの構造と不可分の関係にあり，データ構造の記述はアルゴリズムについての記述と並行して行なわれるべきものである．本書はこのような意図をもって書かれたデータ構造の教科書であり，これが本書の特異なタイトルを説明している．

本書では，上のような理由から，通常のデータ構造の書物に比べると，比較的まとまった大きな例題を選んで説明が行なわれている．そして，それに対するプログラム全体を作成するという立場をとっており，教科書としてはかなり大きなプログラムが載せられている．これらのプログラムは著者の提唱する段階的詳細化法により構成され，初期の抽象的仕様記述から最終的なプログラムが生成される各段階が詳細に記述されている．したがって，本書は単にデータ構造に関する教科書としでだけでなく，プログラミング方法論の教科書としても格好なものであろう．

著者 N. Wirth についての紹介が遅れたが，彼はチューリッヒの連邦工科大学 (ETH) の教授であり，現在では PASCAL 言語の設計者というのが最も通りがよいであろう．PASCAL はもともと教育用として設計されたものであり，本書のプログラムもすべて PASCAL によって記述されている．PASCAL に関する参考書が十分でない現在，本書はそのような目的にも役立つであろう．著者はまた PASCAL 以外にもいくつかのプログラム言語の設計者として有名であり，言語設計におけるセンスと才能には極めて高いものがある．またプロ

グラミング方法論の分野においても活躍しており，段階的詳細化法の提唱者としても知られている．

以上のように，本書は高名な研究者による名者であり，欧米の多数の大学等において教科書として採用されていると聞いているが，これを我国においても利用できればと考えたのが本書の翻訳をお引き受けした理由である．訳者としてはそれなりの努力をしたつもりではあるが，必ずしも満足のゆく出来映えでないのは残念である．これはもちろん訳者の力不足が主な原因であるが，名著の翻訳ということで多少気をつかいすぎ，訳者があまり自由に振舞えなかった点のあったこともお察しいただけたら幸いである．翻訳上の誤りがないように努めたつもりであるが，もし何かしらお気づきの点があれば訳者までお知らせいただきたい．ただ，原著の誤りや印刷ミスをかなり発見して訂正したので，訳の拙いところはその分で埋め合わせていただけたらと思っている．

訳語については，原則として一般に使われているものを使うように心がけた．適当な日本語訳のないものに対しては，特に新しい訳語を作ることはせず片かな書きとした．片かな書きの部分が少ない方が望ましいことには違いないが，これまでに定着したものにとって代わるような無理のない訳語を作るのはかなり難しく，かえって不適当な訳語のために本書の価値が多少とも損なわれるのを懸念したからである．

本書の翻訳で一番苦慮したことは，段階的詳細化法の中間ステップに現われる文章風の長い手続き名やコメントの扱いである．本来ならこれも日本語に直すべきところであるが，本訳では原則としてこれらはそのまま訳さずに残しておき，本文中においてそれが参照されたときに日本語訳を付加するという方針をとった．その理由は，これらの手続き名やコメント中の語がプログラムの中で識別名として使われていることが多く，これを訳すとかえってプログラムが理解しにくくなると判断したからである．また，プログラムの部分は原著のものをそのまま用い，植字の際の誤りの混入を避けたいと考えたこともその理由のひとつである．

最後に，本書の翻訳に際して，訳語の選定や校正に関して訳者の囲りの方々から多大の御援助をいただいたことを深く感謝するものである．

1979年5月

片 山 卓 也

われわれの取り計いにより，中将 L. Euler は次のような陳述をしている．彼は卒直にこう告白している．

Ⅲ．私は数学者の王ではあるが，自分の間違った公式から，重力によって球の中心に引きつけられる物体は，中心においてその方向を急に逆転するであろうという結論を導き，常識や通説を攻撃したことを常に恥しく思っている．

Ⅳ．今後，正しくない公式によって理性が二度とこのように欺かれることがないように，最大の努力をするつもりである：逆説的な結果をうのみにして，“これは現実に反するように見えるかも知れないが，我々は自分の良識より計算の方を信用すべきである”ことを要求したことを深く陳謝するものである．

Ⅴ．これからは，1つの結果を得るために60頁もの計算をするようなことはしないであろう．だれでも，いくらかの注意深い熟慮の後では，10行もあれば結論を得ることができる．また，もし3日3晩かけるような計算をすることになったとしても，どの(計算)原理が最適かと考えるために予め15分は費すつもりである．

ヴォルテール著　“アカキア博士の酷評”（1752年12月）から

目　　次

第1章

基本的データ構造

1.1 はじめに

現在のディジタル計算機は，複雑で時間のかかる計算を高速に実行するために発明されたのであるが，その大部分の応用では，大量の情報を記憶しアクセスする能力がその主要な役目であり，基本的特性と考えられている．そして計算すなわち算術演算を実行する能力には，多くの場合余り関心が払われない．

これらのすべての場合において，処理すべき大量の情報はある意味で現実世界のある部分の**抽象的表現**である．計算機にかけられるデータは，現実世界についてのデータからなるある選ばれた集合，すなわち，そこから必要な結果が導き得ると考えられる集合である．このデータは，当面の問題には本質的でなくかつ無関係であるような性質や特徴が無視されている，という点で現実のある抽象化を表わしており，それゆえ，この抽象化は事実の簡単化でもある．

たとえば，従業員の個人ファイルについて考えてみよう．このファイル上では，すべての従業員は雇用者や彼の会計手続きに関係あるデータの集合によって表現されている．この集合には，たとえば，名前とか給与とか従業員を識別するものが含まれているであろうが，毛髪の色，体重や身長のような無関係な情報は含まれていないであろう．

計算機を使うにせよあるいは使わないにせよ，ある問題を解くには現実をどのような仕方で抽象化するかを選択すること，すなわち現実の状況を表現するデータ集合を定義することが必要である．そしてこの選択は解くべき問題に沿って決めなければならない．次に行なうべきことは，この情報の表現法を選択することであるが，この選択はその問題を解くのに使用される道具すなわち計算機のもっている機能に従って決めなければならない．多くの場合，これらの2つのステップはまったく独立というわけではない．

データ表現の選択はしばしばかなり困難なことであり，また利用可能な機能によって一意に決められるものではなく，そのデータにほどこされる演算を考慮の上で決定しなければならないのが常である．数の表現法はこの良い実例であるが，数自身はまた特徴化すべき対象のもつ性質を抽象化したものである．もし，加算が唯一のあるいは少なくとも主な演算であるならば，数 n を表現する1つの良い方法は，n 本の縦棒を書くことである．この記法に基づく加算の規則は全く明らかで簡単なものである．ローマ数字はこれと同じ原理に基づいており，小さい数に対しては加算の規則は同様に簡単である．一方，アラビア数字では，（小さい数に対しては）明らかとは言いがたい規則を必要とし，記憶が必要である．しかしながら，大きな数の加算や乗除算を考えるときにはこの状況は逆になる．すなわち，位による桁の重みづけに基づいた規則的な構成原理のおかげで，これらの演算はより簡単な演算に分解され，ローマ数字の場合よりよほど簡単になる．

計算機では2進数による内部表現を利用していることは良く知られている．この表現は桁の数が増えるので人間には向いていないが，電子回路には最も適したものである．なぜなら 0，1 という2つの値は電流，電界あるいは磁界の有無によって表現できるからである．

この実例から，しばしば表現の問題は多レベルにわたっていることがわかる．たとえば，物体の位置を表現する問題が与えられたとしよう．このとき，最初の決定は，実数の対でそれを表現する座標（直交座標，極座標等）の選択である．次には，浮動小数点数の表現が決められるが，これによってすべての実数 x は，適当な底に対する仮数 f と指数 e（たとえば $x=f\cdot 2^e$）という2つの整数によって表現される．第3番目の決定では，このデータが計算機に格納されることを考えて，整数の2進数表現が行なわれ，そして最後の決定では，磁気記憶装置中での磁束の向きを利用した2進数の表現法が決定される．明らかに，この一連の決定のうち最初のものは，主に問題の形によって影響されるが，それ以後のものは道具立てとそれに関する技術の進歩にも依存している．したがって，数の表現法や記憶素子特性についての決定を，プログラマに要求することはほとんど不可能である．これらの“低レベルの決定”は計算機装置の設計者にゆだねられるべきである．なぜなら現在の技術について最も多くの情報を持っているのは彼等であり，したがって，数が関係するようなすべての（あるいはほとんどすべての）応用に対して受け入れられるような意味のある決定をすることができるからである．以上より，プログラム言語の重要性が明らかとなる．プログラム言語は，この言語で使用される用語を理解できる抽象的計算

機を表わしているが，それは現実の計算機で扱われる対象を適当なレベルで抽象化し，これを具体化したものである．したがって，このような“高レベル”言語を使用するプログラマは，もし数がこの言語の基本要素として含まれていれば，数の表現の問題から解放されることになる．

データ処理のほとんどの問題に共通で便利な基本機能をもった言語を使用することの重要性は，でき上がったプログラムの信頼性にある．ビット，“語”あるいはジャンプといったものより，数，集合，列あるいは繰返し等の良く知られた概念を用いてプログラムを設計する方がより簡単である．もちろん実際の計算機では，数でも集合でも，あるいは列でも，すべてのデータは多量のビットを用いて表現されるであろう．しかし，抽象化の細部に立ち入る必要がなかったり，あるいは計算機（またはコンパイラ）によって選ばれた表現が適切であるうちは，このような詳細はプログラマにとって無関係である．

抽象化が与えられた計算機に近ければ近いほど，その言語の設計者にとって表現の選択はより簡単であり，そしてある1つの選択が，考えられるすべての（あるいはほとんどすべての）応用に対して適切である可能性が高くなる．この事実は，与えられた現実の計算機からの抽象化の度合いに一定の限界を設けることになる．たとえば，汎用言語に幾何学的対象を基本データ項目として含めることは無意味である．なぜなら，それらに向いた表現は，その固有の複雑さのゆえに，適用される演算に大きく依存するからである．汎用言語やそのコンパイラの設計者は，これらの演算の性質や頻度について知らないので，彼が行なうどのような選択も，応用によっては不適当である可能性がある．

本書ではこれらの点を考慮して，アルゴリズムやデータを記述するための記法を選択した．われわれはビット列のような計算機に依存したものを用いるより，数，列，集合などのように熟知している数学上の概念を用いたい．また同様に，効率のよいコンパイラが作れるような記法を用いたいと思う．その表現が決められないような抽象的記法で計算機のプログラムを書いても何の役にも立たないし，そうかといって，非常に機械と関連したあるいは依存した言語を用いることも利口なことではない．

プログラム言語 PASCAL は，この両極端の妥協案として設計され，本書を通じて使用されている [1-3; 1-5]．この言語はいくつかの計算機上でのインプリメントが成功しており，またその記法が実際の計算機に十分に近いので，その選ばれた特徴や表現を明解に説明できることが示されている．この言語はまた他の言語，特に ALGOL 60 に非常に近いので，本章で習得するだけで十分に使用することができるであろう．

1.2　データ型の概念

数学では，変数をその重要な特徴によって分類するのが普通である．実変数，複素変数，論理変数は明確に区別されるし，また個々の値を表わす変数，値の集合を表わす変数，集合の集合を表わす変数の間にも，さらには関数，汎関数，関数の集合の間などにも明確な区別がなされている．この分類はデータ処理においても等しく重要である．われわれは，**どの定数，変数，式あるいは関数もある1つの型をもつ**，という原則を設定する．この型は，本質的には，定数が属する値の集合，変数や式がとる値の集合，あるいは関数によって生成され得る値の集合を特徴づけるものである．

数学のテキストでは，変数の型は印刷文字の形で区別するのが普通であるが，計算機のプログラムではこれは可能ではない．なぜなら，通常計算機で使える活字は1種類（英文字）しかないからである．したがって，定数，変数，関数の**宣言**においてその型を明示し，かつ，その宣言をそれらの定数，変数，関数の使用に先立った場所（プログラム・テキスト上で）で行なうという規則が広く受け入れられている．コンパイラは計算機記憶装置中でのその表現を決定しなければならないということを考えると，この規則は特に意味のあることである．ある変数の取り得る値の大きさから，その変数に割り当てるべき記憶容量を決定できるので，この情報がコンパイラに知らされていると，いわゆる動的記憶割当てを避けることができる．これはしばしばアルゴリズムの効率的実現の鍵である．

本書において使用され，またプログラム言語 PASCAL で実現されているデータ型の概念の基本的性質は次の通りである [1-2]：

1. データ型は，定数の属する値の集合，変数や式のとる値の集合あるいは関数や演算子によって生成される値の集合を決定する．
2. 定数，変数，式が表わす値の型は，計算プロセスを実行させるまでもなく，その形や宣言から決定することができる．
3. すべての演算子や関数は，その引数や結果の型が定まっている．もし1つの演算子の引数として複数の型が許されるときには（たとえば，＋は整数および実数両方の加算に用いられている），結果の型は言語固有の規則から決定することができる．

この結果，コンパイラは型に関するこの情報を利用して種々の構文の適合性や正当性をチェックすることができるであろう．たとえば，論理値が実数変数

に代入されることは，プログラムの実行をしなくても検出することができる．プログラム・テキスト中のこの種の冗長性は，プログラム作成上非常に有用な助けとなり，またこれは良い高級言語の，機械語やアセンブラ言語に対する基本的利点と考えられなければならない．明らかに，プログラムがデータ型の概念をもつ高レベル言語で作られたものであろうと，あるいは型のないアセンブリ言語で作られたものであろうと，データは究極的には多数のビットによって表現されるであろう．また，計算機にとっては，記憶装置は明確な構造をもたない一様かつ多量なビットの集合である．しかしながら，人間であるプログラマが計算機記憶装置中の単純な“風景”の中で意味を認識できるのは，正にこの抽象的構造によるのである．

本書およびプログラム言語 PASCAL で述べられている理論は，データ型を定義する一定の方法を明示している．たいていの場合，新しいデータ型はそれ以前に定義されたデータ型を用いて定義される．そのような型に属する値は，それ以前に定義された**成分型**に属する**成分値**から構成されるのが普通であり，このような値は**構造をもっている**といわれる．もし，ただ1つの成分型しかないとき，すなわちすべての成分が同じ型のときには，これは**基底型** (base type) として知られているものである．

型 T に属する異なる値の個数は，T の**基数**と呼ばれる．この**基数**は，型 T の変数 x (x:T と表わされる) を表現するのに必要な記憶容量の測度を与える．

成分型は再び構造をもっていても良いので，構造の完全な階層が構成され得る．もちろん，構成の最終的成分は原子的でなければならない．それゆえ，そのような基本的で構造をもたない型を導入するための記法も用意しておく必要がある．直接的な方法は，その型を構成している値を**列挙**することである．たとえば，平面幾何図形に関するプログラムにおいては，識別名“矩形”，“正方形”，“楕円”，“円”によって表わされる値からなる“形”と呼ばれる基本型が導入されるかも知れない．しかし，このようにプログラマが定義した型の他にも，既定義といえる**標準型**がいくつかあるが，これには，**数**や**論理値**が含まれるのが普通である．もし，個々の値の中に順序が存在するならば，その型は順序つきである，または**スカラ**といわれる．PASCAL では構造をもたないすべての型は順序つきである．そして陽に列挙する場合には，その列挙の順番にしたがって値は順序づけられている．

これだけの道具立てがあると，基本型を定義し，任意の深さのネスト構造をもった型を構築することができる．実際には，成分型を結合して構造を作る一般的方法が1つしかないのでは十分でなく，汎用プログラム言語においてはそ

の表現能力と使用法に関する実際問題を正当に考慮すれば，複数個の**構造化の方法**をもっていなければならない．しかし，数学的な意味ではそれらは皆等価であるといってよい．すなわちそれらは，その値を構成したり，その値の成分を選択するために使える演算子が異なっているだけである．ここで述べる構造化の基本的方法は，**配列**，**レコード**，**集合**および**列**（ファイル）である．もっと複雑な型は，通常は“静的な”型としては定義されず，その代わり，プログラムの実行中に動的に生成され，大きさや形が変化する．そのような構造は4章の主題であり，リスト，リング，木や一般の有限グラフが含まれる．

変数やデータ型は計算で使用するためにプログラムに導入されているが，このためには，さらに**演算**の集合がなければならない．データ型についてと同様に，基本的標準演算と，それから合成演算を定義するための構造化手法がいくつか用意されている．演算を合成することは，しばしばプログラミング技術の中心と考えられているが，データの適切な合成も同様に基本的かつ本質的であることが明らかになるであろう．

最も重要な基本演算は，**比較**と**代入**，すなわち等値性（と順序つき型の場合には順序）の判定と等値性を強制するための命令である．本書では，これらの2つの演算の基本的な差異を強調するために，記法上明確な区別をしている．(FORTRAN や PL/I のような広く使われているプログラム言語で，等値記号を代入演算子としても用いており，この区別が不明確なのは残念なことである)．

等値性の判定： $x=y$

x への代入： $x:=y$

これらの基本演算子はすべてのデータ型に対して定義されているが，データが多量で複雑に構造化されているときには，その実行にはそれなりの計算量が必要であることに注意しなければならない．

等値性（または順序）の判定と代入以外に，基本的でありかつ暗黙に定義された演算子のクラスとして，いわゆる**型変換演算子** (transfer operators) がある．これは，データ型からデータ型への写像であり，構造をもったデータ型との関連で特に重要である．構造をもった値は，その成分値からいわゆる**構成子**によって生成され，成分値は**選択子**によって抽出される．構成子や選択子は，成分型を構造をもった型に写像する，またはその逆を行なう型変換演算子である．すべての構造化法にはそれ固有の構成子と選択子があり，それらはその記法が明らかに異なっている．

標準基本データ型には，また標準基本演算子の集合が必要である．したがって，数および論理値という標準データ型に対しては，通常の算術演算および命題論理演算が導入されている．

1.3 基本データ型

多くのプログラムにおいて，たとえ数に関する性質が含まれないときでも，場合の数が少ないときの選択を表現するために整数が用いられる．このような目的のために，新しい，基本的かつ構造をもたないデータ型Tを，そのすべての可能な値 $c_1, c_2, \cdots, c_n$ の集合を列挙することによって導入する．

$$\boxed{\textbf{type}\ T=(c_1, c_2, \cdots, c_n)} \tag{1.1}$$

T の基数（要素の数）を $card(T)$ と書くことにすれば，$card(T)=n$ である．(以後，T の基数を "基数(T)" とも表わす)．

[例]

type *shape* = (*rectangle, square, ellipse, circle*)
type *color* = (*red, yellow, green*)
type *sex* = (*male, female*)
type *Boolean* = (*false, true*)
type *weekday* = (*Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday*)
type *currency* = (*franc, mark, pound, dollar, shilling, lira, guilder, krone, ruble, cruzeiro, yen*)
type *destination* = (*hell, purgatory, heaven*)
type *vehicle* = (*train, bus, automobile, boat, airplane*)
type *rank* = (*private, corporal, sergeant, lieutenant, captain, major, colonel, general*)
type *object* = (*constant, type, variable, procedure, function*)
type *structure* = (*file, array, record, set*)
type *condition* = (*manual, unloaded, parity, skew*)

このようなデータ型の定義には，単に新しい型識別名のみでなく，新しい型に属する値を指示するための識別名の集合も導入する必要がある．これらの識別名は，プログラムの中で定数として使用することができ，プログラムの読み易さをかなり高めることができる．たとえば，変数 s, d, r, b を

$$\textbf{var}\ s : sex$$
$$\textbf{var}\ d : weekday$$
$$\textbf{var}\ r : rank$$
$$\textbf{var}\ b : Boolean$$

のように導入すれば，次のような代入文が可能である．

$$s := male$$
$$d := Sunday$$
$$r := major$$
$$b := true$$

明らかにこれらは，

$$s := 1 \qquad d := 7 \qquad r := 6 \qquad b := 2$$

のように，s, d, r, b を整数として定義し，定数をその列挙の順番によって整数に写像する方法よりもわかりやすいものである．さらに，コンパイラは，たとえば

$$s := s+1$$

のような非数値型に対する軽率な算術演算をチェックすることができる．型が順序つきであるときには，ある値 x の直後および直前の値を作り出す関数を導入することは意味がある．これらの関数は $succ\,(x)$，$pred\,(x)$ によって表わされる．T の値集合の中での順序は，次の規則によって定義される．

$$(c_i < c_j) \equiv (i < j) \tag{1.2}$$

1.4　標準基本型

標準基本型とは，ほとんどの計算機にその機能の一部として組み込まれているような型のことである．それらは，整数，論理値および印刷可能な文字を含んでいる．より大きな計算機では，実数も十分な基本演算の組とともに組み込まれている．これらの型は，次の識別名によって表わされる．

integer,　*boolean*,　*char*,　*real*

型 *integer*（以後，**整数型**という）は整数全体の部分集合より成るが，その大きさは個々の計算機により異なる．しかしながら，この型のデータに対する

どの演算も，通常の算術の規則に対応しており，かつその結果が，表現し得る範囲の大きさを越えたときには，計算が中断されるものと仮定されている．標準演算は加算（+），減算（−），乗算（*）および除算（**div**）の四則演算である．この最後の演算については，結果は整数であり，剰余は無視されるものとする．すなわち正整数 m, n に対して，次が成立する．

$$m-n < (m \operatorname{\mathbf{div}} n) * n \leq m \tag{1.3}$$

剰余演算子は次の方程式により定義される．

$$(m \operatorname{\mathbf{div}} n) * n + (m \operatorname{\mathbf{mod}} n) = m \tag{1.4}$$

したがって，$m \operatorname{\mathbf{div}} n$ は m を n で割ったときの商であり，$m \operatorname{\mathbf{mod}} n$ はその剰余である．

型 *real*（以後，**実数型**という）は実数の部分集合を表わす．整数だけを含む算術演算が正確な結果を生ずると考えられるのに反し，実数型を含む演算では，有限の桁数内での計算から生ずる丸め誤差の範囲内で結果が不正確となることが許されている．これが，ほとんどのプログラム言語で整数型と実数型が明確に区別されている主な理由である．

実数同志の除算は実数の商を生じるが，これは，整数の除算 **div** に対比して，スラッシュ（/）によって表わされる．

型 *boolean*（以後，**論理型**という）に属する2つの値は，識別名 *true* と *false* によって表わされる．論理演算子には，論理和，論理積，否定があるが，これらは**表 1.1** で定義される．

論理和は記号∨（または **or**），論理積は∧（または **and**），否定は¬（または **not**）によってそれぞれ表わされる．比較は論理型の値をその結果として生じるような演算であることに注意してほしい．したがって，比較の結果は変数に代入することもできるし，あるいは，論理式において論理演算子のオペランドとして用いることもできる．たとえば論理変数 p, q および整数変数 $x=5$,

表 1.1 論理演算子

p	q	$p \vee q$	$p \wedge q$	$\neg p$
true	*true*	*true*	*true*	*false*
true	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*
false	*false*	*false*	*false*	*true*

$y=8$, $z=10$ が与えられたとき，2つの代入文

$$p := x = y$$
$$q := (x < y) \wedge (y \leq z)$$

によって，$p = false$, $q = true$ となる．

標準型 *char*（以後，**文字型**という）は印刷可能な文字集合から構成される．残念なことにすべての計算機で使用され，一般的に認められた文字集合は存在しない．したがって，"標準"という言葉は正しくはなく，あるプログラムが実行される計算機システムで"標準"という意味に解釈すべきである．

文字集合は国際標準化機構（ISO）によって定められており，特にそのアメリカ版 ASCII (American Standard Code for Information Interchange) は，最も広く受け入れられている文字集合であろう．ASCII 集合は補遺 I に載せてある．それには，95 の印刷文字と 33 の制御文字が含まれているが，後者は，主にデータ伝送と印刷装置の制御のために用いられるものである．64 の印刷文字からなる部分集合は制限 ASCII 集合と呼ばれ，広く用いられている．

文字（すなわち文字型の値）を含んだアルゴリズムが，計算機システムとは独立に設計できるために，文字集合は次のようになっていると仮定しておきたい：

1. 文字型は 26 個の**英字**と 10 個の**数字**および句読点のような特殊文字を含んでいる．
2. 英字と数字には，次のような一貫した順序がついている．

$$\begin{aligned} (\text{'A'} \leq x) \wedge (x \leq \text{'Z'}) &\equiv x \text{ は英字である} \\ (\text{'0'} \leq x) \wedge (x \leq \text{'9'}) &\equiv x \text{ は数字である} \end{aligned} \qquad (1.5)$$

3. 文字型は，印刷されない空白文字を含んでいるが，これは区切り記号に使うことができる．**図 1.1** では，空白文字は␣によって表わされている．

文字型と整数型間の2つの型変換演算子は，機械独立な形のプログラムを書

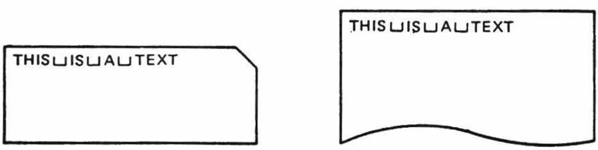


図 1.1　テキストの表現

く上で特に重要である．それらは，集合 *char* 中における文字 c の順序数を与える $ord(c)$ と，集合 *char* 中の i 番目の文字を与える $chr(i)$ である．したがって *chr* と *ord* は互に逆関数になっており，次式が成立する．

$$\begin{aligned} &ord(chr(i)) = i \quad [chr(i) \text{ が定義されている場合}] \\ &chr(ord(c)) = c \end{aligned} \tag{1.6}$$

特に良く知られている関数は，

$$\begin{aligned} &f(c) = ord(c) - ord(\text{'0'}) = \text{数字中の } c \text{ の位置} \\ &g(i) = chr(i + ord(\text{'0'})) = i \text{ 番目の数字} \end{aligned} \tag{1.7}$$

である．たとえば $f(\text{'3'})=3, g(5)=\text{'5'}$ である．f, g は互いに逆関数の関係にある．
すなわち

$$\begin{aligned} &f(g(i)) = i \qquad (0 \le i \le 9) \\ &g(f(c)) = c \qquad (\text{'0'} \le c \le \text{'9'}) \end{aligned} \tag{1.8}$$

これらの型変換関数は，数の内部表現を数字列に変換したり，その逆の変換を行なうために用いられる．事実，これらの関数は，そのような変換を1桁という最も初等的な場合について表現したものである．

1.5 区　間　型

変数がある型の特定の区間内の値をとるということはよくある．これは，次の形式の区間型の変数を定義することによって表現することができる．

$$\boxed{\textbf{type } T = min\,..\,max} \tag{1.9}$$

ここで，*min* と *max* は区間の下限値と上限値である．

[例]

type *year*=1900..1999
type *letter*='A'..'Z'
type *digit*='0'..'9'
type *officer*=*lieutenant*..*general*

変数が

$$\textbf{var}\ y : \textit{year}$$
$$\textbf{var}\ L : \textit{letter}$$

のように与えられたとき，$y := 1973$ および $L := \text{'}W\text{'}$ は許されるが，$y := 1291$ や $L := \text{'}9\text{'}$ は許されない．このような代入文の正当性は，代入すべき値が定数や同じ型の変数の時には，コンパイラによって検証することができる．しかしながら i を整数型，c を文字型の変数とするとき，

$$y := i \quad \text{および} \quad L := c$$

の形の代入文が正当なものであるか否かの判定は，プログラムの実行によってのみ可能である．実際には，このような判定を行なうシステムは，プログラム作成上きわめて価値のあるものであることがわかっている．誤りの検出にこのような冗長な情報を利用できることが，高級言語を使用する基本的動機の１つである．

1.6 配 列 構 造

配列は，FORTRAN や ALGOL 60 を含む多くのプログラム言語では仕様上許される唯一の構造であり，したがって，多分最も広く知られたデータ構造であろう．配列は**一様な**構造である．すなわち，それは**基底型**と呼ばれる同じ型の成分†から構成されている．配列はまた，いわゆる**ランダム・アクセス構造**でもある．すなわち，すべての成分はランダムに，また同等に選択することができる．個々の成分は，構造全体の名前にいわゆる**インデックス**を付加したものによって指定される．このインデックスは，配列の**インデックス型**として定義された型の値でなければならない．したがって，配列型 T の定義では，基底型 T_0 とインデックス型 I を指定することになる．

$$\boxed{\textbf{type}\ T = \textbf{array}\,[I]\ \textbf{of}\ T_0} \qquad (1.10)$$

[例]

$$\textbf{type}\ \textit{Row} = \textbf{array}\,[1\,..\,5]\ \textbf{of}\ \textit{real}$$
$$\textbf{type}\ \textit{Card} = \textbf{array}\,[1\,..\,80]\ \textbf{of}\ \textit{char}$$
$$\textbf{type}\ \textit{alfa} = \textbf{array}\,[1\,..\,10]\ \textbf{of}\ \textit{char}$$

† **訳注**　通常配列のメンバーは要素（element）といわれることが多い．本書では，原則的に順序のついた集合のメンバーを成分（component），順序のない集合のメンバーを要素と使い分けている．以後の部分では多少の混同はあるが，element は要素，component は成分と訳してある．

宣言

$$\mathbf{var}\ x : Row$$

によって定義され，その成分値が $x_i=2^{-i}$ を満たす変数 x は**図 1.2** のように表わすことができるであろう．

型がTで成分値が $c_1, \cdots, c_n$ の構造値 x は，**配列構成子**および代入文によって次のように表わすことができる．

$$x := T(c_1,\ \cdots,\ c_n) \tag{1.11}$$

構成子の逆演算子は**選択子**であり，その配列から個々の成分を選び出す．配列変数が与えられたとき，配列選択子は配列名のあとに成分インデックスをつけて表わす．

$$\boxed{x[i]} \tag{1.12}$$

配列，特に大きな配列を扱う場合には，1つの成分を選び，それを更新するという扱いをするのが普通であり，全く新しい構造値を構成しなおすことは余り行なわない．これは次のように，選ばれた成分変数に対して代入を行なうことによって表現できる．

[例]

$$x[i] := 0.125$$

選択的更新では，もちろん1つの成分のみが変化するわけであるが，概念的には構造値全体も変化したと考えなければならない．

配列インデックスすなわち配列成分の“名前”が確定した（スカラ）データ型をもたなければならないということには非常に重要な意味がある．インデッ

x_1	0.5
x_2	0.25
x_3	0.125
x_4	0.0625
x_5	0.03125

図 1.2　型 *Row* の配列

クスは計算値でもよい．すなわち，インデックス定数の代わりにインデックス**式**を書いてもよい．このときには，この式が評価されて，その結果として選択すべき成分が決定される．このような一般化は，非常に重要で強力なプログラミングの機能ではあるが，同時に最も頻繁に起こるプログラミング上の誤りの原因でもある．なぜなら，インデックス式の値がその配列のインデックスの範囲からはずれてしまう可能性があるからである．適切な計算機システムでは，このような実在しない配列成分への誤ったアクセスに対して警告を出すべきであろう．

通常インデックス型はスカラ型，すなわち順序関係の定義されている非構造型である．もし，配列の基底型もまた順序つき型であるときには，そのような配列型の上に自然な順序関係が導入される．2つの配列間の自然な順序は，対応する2つの成分が等しくないような最小のインデックスによって決定される．これは形式的には次のように表現できる：

2つの配列 x, y が与えられたとき，関係 $x<y$ が成立するのは，次を満たすインデックス k が存在する場合であり，またその場合に限る．$x[k]<y[k]$ かつすべての $i<k$ に対して，$x[i]=y[i]$　(1.13)

たとえば，

$$(2,3,5,7,9) < (2,3,5,7,11)$$
$$\text{'LABEL'} < \text{'LIBEL'}$$

しかしながら，ほとんどの応用では，配列型の上に順序が想定されることはない．

構造型の基数は，その成分型の基数の積である．配列型Aの成分はすべて基底型Bであるので，次の関係式を得る．

$$\text{基数}(A) = (\text{基数}(B))^n \quad (1.14)$$

ただし，n=基数(I) かつ I はこの配列のインデックス型である．

次の短いプログラムは，配列選択子の使い方を示している．このプログラムの目的は，値 x をもつ成分のうちで，最小のインデックスのものを見つけることである．探索は配列 a を順に走査することにより行なわれる．

```
var a : array[1..N] of T;  {N>0}
i :=0;
repeat i :=i+1 until(a[i]=x) ∧ (i=N);          (1.15)
if a[i] ≠ x then "there is no such element in a"
```

このプログラムの1つの変形として，配列の最後に“番人”(sentinel) を置くというよく知られた手法を用いるものがある．番人の目的は，繰返しの終了条件を簡単化することである．

$$\begin{aligned}&\textbf{var}\ a:\textbf{array}[1\,..\,N+1]\ \textbf{of}\ T;\\&i:=0;\ a[N+1]:=x;\\&\textbf{repeat}\ i:=i+1\ \textbf{until}\ a[i]=x;\\&\textbf{if}\ i>N\ \textbf{then}\ \text{“\textit{there is no such element in a}”}\end{aligned} \tag{1.16}$$

代入 $a[N+1]:=x$ は，**選択的更新**すなわち構造をもった変数の特定な成分の変更の例になっている．文 $i:=i+1$ が何回繰り返し実行されようとも成立する本質的な条件は，プログラム (1.15), (1.16) ともに

$$a[j]\neq x,\ \text{ただし}\ j=1,\cdots,i-1$$

である．この条件は，それゆえに**ループ不変式**と呼ばれる．

探索走査は，もちろん要素が既に順序づけられている（ソートされている）ならば，かなり高速化できる．この場合，探索すべき区間を繰り返し半分にしていくという原理が最も普通のものであろう．これは，**2分割法**または**2分探索**と呼ばれ，(1.17) で示される．各繰返しにおいて，インデックス i,j 間の検査区間は2分される．したがって，必要となる比較回数はたかだか $\log_2(N)$ である．

$$\begin{aligned}&i:=1;\ j:=N;\\&\textbf{repeat}\ k:=(i+j)\ \textbf{div}\ 2;\\&\qquad \textbf{if}\ x>a[k]\ \textbf{then}\ i:=k+1\ \textbf{else}\ j:=k-1\\&\textbf{until}\ (a[k]=x)\vee(i>j)\end{aligned} \tag{1.17}$$

(繰返し文の入口での関連した不変式は

$$\begin{aligned}&a[h]<x,\ \text{ただし}\ h=1,\ \cdots,\ i-1\\&a[h]>x,\ \text{ただし}\ h=j+1,\ \cdots,\ N\end{aligned}$$

である．したがって，プログラムが $i>j$ として終了したなら，これは $a[h]=x$ かつ $1\leq h\leq N$ を満たすものが存在しないことを意味する．)

配列型の成分はそれ自身構造をもっていてもよい．その成分が再び配列であるような配列は**行列**と呼ばれる．たとえば

$$M:\textbf{array}[1\,..\,10]\ \textbf{of}\ \textit{Row}$$

は，10 個の成分（行）から成る配列であり，その各成分は5個の実数型の成分から構成されている．そして，これは実数成分をもつ 10×5 行列と呼ばれる．選択子を

$$M[i][j]$$

のように連接することにより，M の i 番目の成分である行 $M[i]$ の j 番目の成分を表わすが，これは，

$$M[i,j]$$

のように短縮する．同様の精神で，宣言

$$M: \textbf{array}[1..10] \textbf{ of array}[1..5] \textbf{ of } \mathit{real}$$

は次のようにもっと簡潔に書くことができる．

$$M: \textbf{array}[1..10,\ 1..5] \textbf{ of } \mathit{real}.$$

もし，配列のすべての成分，または連続した成分よりなる部分にある一定の演算をほどこすときには，次の例で示されるように，**for** 文を使ってこの事実を強調することができる．
小数 f が配列 d によって，

$$f=\sum_{i=1}^{k-1} d_i * 10^{-i}$$

のように $k-1$ 桁の 10 進表現によって表現されていると仮定しよう．f を2で割ることを考える．これは，すべての $k-1$ 個の桁 d_i を割るという良く知られた操作を，$i=1$ から始めて繰り返し行なえばよい．すなわち，前の桁からの繰り下りを考慮しながら各桁を2で割り，その剰余を次のステップのために保持するのである［(1.18) 参照］．

$$\begin{aligned} &r:=10*r+d[i];\\ &d[i]:=r \textbf{ div } 2;\\ &r:=r-2*d[i] \end{aligned} \qquad (1.18)$$

プログラム 1.1 では，このプロセスを利用して2の負べきの表を計算している．$2^{-1}, 2^{-2}, \cdots, 2^{-n}$ を計算するために行なう2分割の繰返しは，再び **for** 文によって表現され，したがって，全体としては2つの **for** 文の入れ子構造になる．

```
program power (output);
{decimal representation of negative powers of 2}
const n = 10;
type digit = 0 .. 9;
var i,k,r: integer;
    d: array [1 .. n] of digit;
begin  for k := 1 to n do
    begin write('.'); r := 0;
        for i := 1 to k-1 do
        begin r := 10*r+d[i]; d[i] := r div 2;
              r := r-2*d[i]; write(chr(d[i]+ord('0')))
        end ;
        d[k] := 5; writeln('5')
    end
end
```

プログラム 1.1 2のべき乗の計算

$n=10$ に対する結果は次のようになる.

```
.5
.25
.125
.0625
.03125
.015625
.0078125
.00390625
.001953125
.0009765625
```

1.7 レコード構造

構造型を作る最も一般的な方法は，任意の要素（それ自身が再び構造型である場合を含めて）を集めた複合型を作ることである．数学からの例としては，2つの実数から作られる複素数や，座標軸によって張られる空間において，その次元から定まる2つ以上の実数から構成される点座標などがある．情報処理での実例としては，氏名，生年月日，性および未既婚の区別のような適切な特性量によって人を記述する方法などがある．

数学では，そのような複合型は，その成分型から構成される**直積**と呼ばれている．これは，この複合型によって定義される値の集合が，その成分型の定める集合からの値の組のすべてから構成されているという事実によっている．し

たがって，**n 組**（*n-tuple*）とも呼ばれるそのような組の総数は，各成分型に属する要素数の積である．すなわち複合型の基数は，成分型の基数の積である．

データ処理では，人や物の記述のような複合型は，通常ファイルや“データバンク”に関連して生じ，その人や物に関連した特性量を記録する．**レコード**という語はそれゆえに，このような複合データの記述に広く用いられており，本書でも直積の代わりにこの用語を用いることにする．

一般に，レコード型 T は次のように定義される．

```
type T = record s1: T1;
                s2: T2;
                 ...
                sn: Tn
         end
```
(1.19)

$$基数(T)=基数(T_1)*\cdots*基数(T_n)$$

[例]

```
type Complex = record re: real;
                      im: real
               end

type Date = record day: 1..31;
                 month: 1..12;
                  year: 1..2000
            end

type Person = record name: alfa;
                     firstname: alfa;
                     birthdate: Date;
                     sex: (male, female);
                     marstatus: (single, married,
                        widowed, divorced)
              end
```

型 T の値は，**レコード構成子**によって構成され，その型の変数に代入される．

$$x:=T(x_1, x_2, \cdots, x_n) \tag{1.20}$$

ここで，各 x_i は型 T_i の値である．

レコード変数

$$z : Complex$$
$$d : Date$$
$$p : Person$$

が与えられたとき，たとえば**図 1.3** のようにして特定の値を代入することができる．

$z := Complex\,(1.0, -1.0)$
$d := Date\,(1,4,1973)$
$p := Person$ ('WIRTH', 'CHRIS', *Date* (18,1,1966), *male*, *single*)

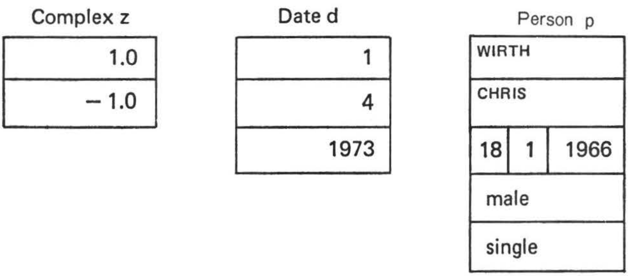


図 1.3 型 *Complex, Date, Person* のレコード

レコード型の定義で導入された識別名 $s_1, \cdots, s_n$ は，その型の変数の各成分に与えられた名前であり，レコード構造変数に適用される**レコード選択子**において使用される．変数 $x : T$ が与えられたとき，その i 番目の成分は次のように表わされる．

$$\boxed{x.s_i} \tag{1.21}$$

x の選択的更新は，この選択子表現を左辺とする代入文によって行なわれる：

$$x.s_i := x_i$$

ここで，x_i は型 T_i の値（式）である．

レコード変数

$$z : Complex$$
$$d : Date$$
$$p : Person$$

が与えられたとき，次のものは z, d, p の成分選択子である．

$z.im$	(実数型)
$d.month$	(型 1..12)
$p.name$	(型 $alfa$)
$p.birthdate$	(型 $Date$)
$p.birthdate.day$	(型 1..31)

型 *Person* の例は，レコード型の成分が再び構造をもっていてもよいことを示している．このように選択子は連続して用いてもよい．異なった構造型が入れ子の形で使用されてもよいことはもちろんである．たとえば，レコード変数 r の成分である配列 a の i 番目の成分は，

$$r.a[i]$$

によって表わされ，配列 a の i 番目の成分（レード構造をしているとする）の選択子 s をもつ成分は，次のように表わされる．

$$a[i].s$$

成分型の要素の組合わせをすべて含むのが直積の特徴である．しかし，実際の応用では，それらのすべてが“合法的”，すなわち意味あるものとは限らないことに注意する必要がある．たとえば，上で定義された型 *Date* は，値

$$(31, 4, 1973) \text{ および } (29, 2, 1815)$$

を含んでいるが，これは実際の日付としては絶対に起こらないものである．したがって，この型の定義は実世界の状況を反映したものではないが，実際の目的には十分であり，プログラムの実行中に意味のない値が生じないようにするのはプログラマの責任である．

次に挙げるプログラムの一部分は，レコード型変数の用法を示している．その目的は，配列変数 a で表わされる“*Person*”のうちで，*female*（女性）かつ *single*（独身）である人数を数えることである：

```
var a: array[1 .. N] of Person;
    count: integer;
count := 0;                                                    (1.22)
for i := 1 to N do
    if (a[i].sex = female) ∧ (a[i].marstatus = single) then
        count := count+1
```

この場合のループ不変式は

$$count = C(i)$$

である．ここで $C(i)$ は，部分集合 $a_1, \cdots, a_i$ 中で *female* かつ *single* であるものの数である．

上の文は，**with 文**によって次のように書くこともできる：

```
for i := 1 to N do
    with a[i] do
        if (sex = female) ∧ (marstatus = single) then
            count := count+1
```
(1.23)

with *r* **do** *s* の意味は，文 *s* の中では，変数 *r* のデータ型の選択子名は *r* を前につけずに用いることができるが，これは変数 *r* を参照するものと解釈されるということである．このwith文はプログラム・テキストを短縮するだけでなく，インデックスされた成分 $a[i]$ の記憶番地を何回も評価することを防ぐものである．

もう1つの例では，(たぶん探索をより高速に行なうために) あるグループの人々は配列 a の中で互いにリンクされていると仮定する．リンク情報は，レコード構造 *Person* 中に新たな成分 *link* を付加することによって表現されている．リンクはレコードを直線的鎖状に結合しており，個々の人の直前，直後の人が容易に見つけられるようになっている．このリンク手法の興味ある性質は，レコード中に格納されている1つの数を利用することにより，鎖が両方向にたどれることである．この手法は次のようなものである．

この鎖の隣合う3つのインデックスを i_{k-1}, i_k, i_{k+1} としよう．k 番目の要素のリンク値は $i_{k+1} - i_{k-1}$ に選ばれている．前向きに鎖をたどるときには，i_{k+1} は2つのインデックス変数の現在値 $x = i_{k-1}$, $y = i_k$ から

$$i_{k+1} = x + a[y].link,$$

によって決定され，後向きにたどる際の i_{k-1} は，$x = i_{k+1}$, $y = i_k$ より

$$i_{k-1} = x - a[y].link$$

によって決定される．

表1.2 は表中のすべての人を性別によってリンクした例である．

レコード構造と配列構造には“ランダム・アクセス”という共通の性質がある．しかし，レコードは，そのすべての成分が同一の型であることが要求されない点でより一般的である．一方，配列は，成分選択子が計算できる値であっ

表 1.2　型 *Person* の成分をもつ配列

	名　前	性　別	リンク
1	Carolyn	F	2
2	Chris	M	2
3	Tina	F	5
4	Robert	M	3
5	Jonathan	M	3
6	Jennifer	F	5
7	Raytheon	M	5
8	Mary	F	3
9	Anne	F	1
10	Mathias	M	3

てもよいので非常に柔軟性があるが，レコードでは，レコード成分選択子はそのレコードの定義において宣言された固定した識別名でなければならない．

1.8　可変レコード構造

実際には，2つのデータ型が単に同じ型の変形であると考えた方が便利であり，かつ自然な場合がよくある．たとえば，前節の型"座標"(*Coordinate*)は，それぞれ(*a*) 2つの長さ，(*b*) 長さと角度，からなる"直交(*Cartesian*)座標"，"極(*polar*)座標"という2つの変形の和であると考えることができる．変数が実際にとる変形を同定するために，第3番目の成分を導入する．これは，**型判別子**または**タグ・フィールド**と呼ばれる．

type *Coordinate* =
　　record case *kind*: (*Cartesian*, *polar*) **of**
　　　　　　Cartesian: (*x*, *y*: *real*);
　　　　　　polar: (*r*: *real*; *φ*: *real*)
　　end

ここで，タグ・フィールドの名前は *kind* であり，座標の名前は，*Cartesian* の場合は x, y，また *polar* の場合は r, φ である．

この型 *Coordinate* の値の集合は，2つの型

$$T_1 = (x, y : real)$$
$$T_2 = (r : real;\ \varphi : angle)$$

のそれの和集合であり，したがってその基数は T_1 と T_2 の基数の和である．

合併すべき2つの型が全く異なってはいずに，部分的には同一の成分を共有

しているような場合が非常によくある．これは**可変レコード**という言葉を生じさせた最も主要な状況である．この例としては前節で定義された *Person* があるが，そこではファイルに記録すべき特性量が，*person*（人）の *sex*（性）によって異なっている．たとえば *male*（男性）に対しては，*weight*（体重），*bearded*（ひげが生えているか否か）が適切と考えられる状況もあるが，*female*（女性）に対しては，3つの特性測定値†が意味があろう（それに反し，体重は秘密であろう）．
次の型宣言はこのような考察から生じたものである．

```
type Person =
    record name, firstname: alfa;
           birthdate: Date;
           marstatus: (single, married, widowed, divorced);
      case sex: (male, female) of
      male: (weight: real;
             bearded: Boolean);
      female: (size: array[1 . . 3] of integer)
    end
```

可変レコード型の宣言の一般形は次のようである．

$$
\begin{array}{l}
\textbf{type}\ T\ =\\
\quad\textbf{record}\ s_1 : T_1; \ldots; s_{n-1} : T_{n-1};\\
\quad\quad\textbf{case}\ s_n : T_n\ \textbf{of}\\
\quad\quad c_1 : (s_{1,1} : T_{1,1}; \ldots; s_{1,n_1} : T_{1,n_1});\\
\quad\quad\quad\cdots\cdots\cdots\cdots\\
\quad\quad c_m : (s_{m,1} : T_{m,1}; \ldots; s_{m,n_m} : T_{m,n_m})\\
\quad\textbf{end}
\end{array}
\tag{1.24}
$$

ここで s_i, s_{ij} は，成分型 T_i, T_{ij} の成分に対する選択子名であり，s_n は型が T_n の判別用タグ・フィールドの名前である．定数 $c_1, \cdots, c_m$ は（スカラ）型 T_n を表わす．型 T の変数 x は，$x.s_n=c_k$ の場合およびその場合に限って，次の成分から構成される．

$$x.s_1,\ x.s_2,\ \cdots,\ x.s_n,\ x.s_{k,1},\ \cdots,\ x.s_{k,n_k}$$

成分 $x.s_1, \cdots, x.s_n$ は m 個の変形に対して共通な部分である．

---

† **訳注**　bust, waist, hip のサイズのことか？

したがって $x.s_n \neq c_k$ であるときに，成分選択子 $x.s_{k,h}$ $(1 \leq h \leq n_k)$ を用いることは，プログラミング上の重大な誤りと考えられなければならず，（上で定義した型 *Person* に関していえば）女性にひげの有無を尋ねたり，あるいは（選択的更新の場合には）彼女にひげを生やすように命ずることに相当する！

このように可変レコード型を使用する場合には，最高の注意が要求されるので，個々の変形データに対する演算は，いわゆる **case 文**という特別の文にまとめて行なうのが最も良い．この文の構造は，可変レコード型の宣言を反映させたものになっている．

$$
\begin{array}{l}
\textbf{case } x.s_n \textbf{ of} \\
\quad c_1 : S_1; \\
\quad c_2 : S_2; \\
\quad \cdots \\
\quad c_m : S_m \\
\textbf{end}
\end{array}
\tag{1.25}
$$

S_k は x が変形 k をとる場合のための文である．すなわち，タグ・フィールド $x.s_n$ が値 c_k をもつとき，そのときに限り選択され実行される．したがって，各々の S_k に含まれる選択子が

$$x.s_1,\ \cdots,\ x.s_{n-1}$$

および

$$x.s_{k,1},\ \cdots,\ x.s_{k,n_k}$$

のみであることを確認することによって，間違った選択子名の使用を避けることは全く容易である．

次の短いプログラムは，可変レコード型 *Coordinate* の変数 a, b によって与えられる2点A，B間の距離を計算するためのものである．計算過程は，直交および極座標の可能な4つの組合わせに対応して異なっている（図1.4参照）．

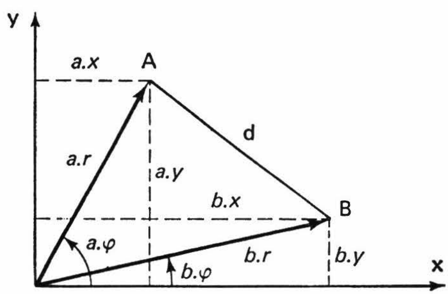


図 1.4　直交および極座標

```
case a.kind of
   Cartesian: case b.kind of
              Cartesian: d := sqrt(sqr(a.x−b.x)+sqr(a.y−b.y));
              Polar:     d := sqrt(sqr(a.x−b.r*cos(b.φ)
                                  +sqr(a.y−b.r*sin(b.φ))
              end;
   Polar:     case b.kind of
              Cartesian: d := sqrt(sqr(a.r*cos(a.φ)−b.x)
                                  +sqr(a.r*sin(a.φ)−b.y));
              Polar:     d := sqrt(sqr(a.r)+sqr(b.r)
                                  −2*a.r*b.r*cos(a.φ−b.φ))
              end
end
```

1.9 集合構造

配列，レコードについで3番目の基本的データ構造は，**集合構造**である．これは次の形の宣言によって定義される．

$$\boxed{\textbf{type } T = \textbf{set of } T_0} \tag{1.26}$$

型 T の変数のとり得る値は，型 T_0 の要素からなる集合である．集合 T_0 のすべての部分集合からなる集合は，T_0 の**べき集合**と呼ばれる．したがって，型 T はその**基底型** T_0 のべき集合を意味する．

［例］

type *intset* = **set of** 0 . . 30
type *charset* = **set of** *char*
type *tapestatus* = **set of** *exception*

2番目の例は，型 *char* によって表わされる標準文字集合に基づいたものであり，3番目のものは，たぶん

type *exception* = (*unloaded*, *manual*, *parity*, *skew*)

のようにスカラ型として定義され，磁気テープ装置のとり得る4つの例外状態の集合に基づいている．変数

$$\begin{aligned}&is : intset\\&cs : charset\\&t\ : \textbf{array}[1..6]\ \textbf{of}\ tapestatus\end{aligned}$$

が与えられたとき，集合型の個々の値は，たとえば次のようにして構成され，代入される[†]:

$$\begin{aligned}&is := [1, 4, 9, 16, 25]\\&cs := [\text{'+'}, \text{'-'}, \text{'*'}, \text{'/'}]\\&t[3] := [mamual]\\&t[5] := [\ ]\\&t[6] := [unloaded .. skew]\end{aligned}$$

ここで，$t[3]$ に代入される値はただ1つの要素 *manual* から成る1要素集合である．$t[5]$ には空集合 [ ] が代入されているが，これは5番目のテープ装置が動作状態（非例外状態）に戻ったことを意味している．一方，$t[6]$ には，4つの例外すべてからなる集合が代入されている．

集合型 T の基数は，

$$基数(T) = 2^{基数(T_0)} \tag{1.27}$$

によって与えられる．これは，T_0 のどの 基数(T_0) 個の要素も“有”，“無”の2つの値のどちらかで表現され，そしてすべての要素は互いに独立であるという事実から容易に導かれる．効率的で経済的な実現のためには，基底型が有限というだけでなく，その基数が十分に小さいということが明らかに重要である．

次の基本演算がすべての集合型に対して定義されている．

*	共通集合
+	和集合
−	差集合
in	要素判定

2つの集合の共通集合や和集合を作ることは，しばしば**集合積**，**集合和**と呼ばれる；集合演算の優先順位は，共通集合演算子が一番高く，次に和・差集合演算子，そして最後が関係演算子である要素判定演算子の順になっている．次の

† 通常の記法とは異なり，集合には {, } の代わりに [,] を用いる．{, } はプログラム中でコメントを囲むためにとっておく．

例は，集合式とそれを完全に括弧でくくった形の等価な式を示している．

$$r*s+t=(r*s)+t$$
$$r-s*t=r-(s*t)$$
$$r-s+t=(r-s)+t$$
$$x\ \mathbf{in}\ s+t=x\ \mathbf{in}\ (s+t)$$

集合構造の応用に関する最初の例題として，コンパイラの簡単な走査プログラムについて述べる．この走査プログラムの目的は，文字列をコンパイルされるべき言語のテキスト単位あるいは，いわゆる語句または**記号**の列に翻訳するものと仮定しよう．走査プログラムは，呼び出されるごとに十分な数の入力文字を読み，1つの出力記号を発生するような手続きとして表現されるものとする．翻訳に関する規則の細目は次の通りである：

1. 出力記号集合は，*identifier*, *number*, *lessequal*, *greaterequal*, *becomes*, および＋，－，＊等の単一の文字に対応する記号から成っている．
2. 記号 *identifier* は，英字で始まる英数字列を読んだときに発生する．
3. 記号 *number* は，数字列を読んだときに発生する．
4. 記号 *lessequal*, *greaterequal* および *becomes* は，それぞれ文字対＜＝，＞＝，：＝を読んだときに発生する．
5. 空白および行の終わりは読みとばす．

入力文字列から次の文字を取り出し，それを変数 x に代入する手続き *read* (x) が用意されていると仮定する．結果の出力記号は大域変数 *sym* に代入される．さらに大域変数 *id*, *num* があるが，その目的は**プログラム 1.2** から明らかであろう．*ch* は入力文字列中の現在走査している文字を表わしている．S は文字から記号への対応を与えている．すなわち，英字や数字以外の文字からなるインデックス領域を持ち，記号をその成分とする配列である．入力文字集合の使い方から，この走査プログラムは，対象としている文字集合中における文字の順序とは独立に作られていることがわかるであろう．

2番目の例は授業時間割についてのものである．M 人の生徒が N 個の**科目** (course) から選択をすると仮定しよう．このとき，いくつかの科目は同時に行なうものとし，矛盾のないように時間割を作らなければならない[1-1].

一般に，時間割の作成は最も困難な組合わせ問題の1つであり，多くの制約条件のもとでいろいろな事柄を考慮して決めなければならない．この例題では，現実の時間割問題に対する解を求めることは考えずに，問題を非常に簡単化しよう．

プログラム 1.2　走査プログラム

```
var ch: char;
    sym: symbol;
    num: integer;
    id: record
            k: 0 .. maxk;
            a: array [1 .. maxk] of char
        end ;
procedure scanner;
   var ch1: char;
begin {skip blanks}
   while ch = '␣' do read(ch);
   if ch in ['A' .. 'Z'] then
      with id do
      begin sym := identifier; k := 0;
         repeat if k < maxk then
                  begin k := k+1; a[k] := ch
                  end ;
            read(ch)
         until ¬(ch in ['A' .. 'Z' , '0' .. '9'])
      end else
   if ch in ['0' .. '9'] then
      begin sym := number; num := 0;
         repeat num := 10*num+ord(ch)-ord('0');
            read(ch)
         until ¬(ch in ['0' .. '9'])
      end else
   if ch in ['<', ':', '>'] then
      begin ch1 := ch; read(ch);
         if ch = '=' then
         begin
            if ch1 = '<' then sym := leq else
            if ch1 = '>' then sym := geq else sym := becomes;
            read(ch)
         end
         else sym := S[ch1]
      end else
   begin {other symbols}
      sym := S[ch]; read(ch)
   end
end {scanner}
```

まず第1にわかることは，並行して実施してもよい科目を見つけるには，個々の学生の**科目登録** (course registration) から得られるデータに基づいて決定を行なわなければならないことである．なぜなら，それには同時には実施できない科目が列挙されているからである．したがって，まず以下のような宣言，および学生を1からMで，また科目を1からNで番号づけるという規約のもとに，データ単純化の過程のプログラムを作る．

```
type course    = 1 .. N;
     student   = 1 .. M;
     selection = set of course;
var s: course;
    i: student;
    registration: array[student] of selection;
    conflict: array[course] of selection;
{Determine the sets of conflicting courses from the individual       (1.28)
 student's course registrations}
for s := 1 to N do  conflict[s] := [ ];
for i := 1 to M do
   for s := 1 to N do
   if s in registration[i] then
      conflict[s] := conflict[s] + registration[i]
```

(このアルゴリズムでは，s 自身 *conflict*[s] に属してしまうことに注意せよ)

さて，主な仕事は，時間割すなわち**かち合う** (conflict) ことがない科目からなる**セッション** (同じ時間に行なわれる科目の集合：session) のリストを構成することである．科目の全集合から，かち合わない科目の適当な部分集合を選び，これを *remaining* という名前の変数から差し引く．そして**残っている** (remaining) 科目の集合が空になるまでこれを続ける．

```
var k: integer;
    remaining, session: selection;
    timetable: array[1 .. N] of selection;
k := 0; remaining := [1 .. N];
while remaining ≠ [ ] do                                              (1.29)
  begin session := next suitable selection;
        remaining := remaining − session;
        k := k+1; timetable[k] := session
  end
```

"next suitable selection"（次の適切な選択）はどのようにして行なったら良いであろうか．最初は残っている科目の中から任意のものを1つ選べばよい．それ以後は，次の候補の選択を，残っている科目で最初に選んだものとかち合わないものの集合からに制限すればよい．この集合を *trailset* と呼ぶ．*trailset* からの候補について調べるときには，既に選択された科目の集合と，この候補のかち合い集合の共通集合が空になるか否かによって決めればよい．このように考えると，文 *"session* :=*next suitable selection"* は次のように詳細化される．

```
var s,t: course;
    trialset: selection;
begin s := 1;
      while ¬(s in remaining) do s := s+1;
      session := [s];  trialset := remaining − conflict[s];
      for t := 1 to N do                                      (1.30)
          if t in trialset then
          begin if conflict[t] * session = [ ] then
                    session := session + [t]
          end
end
```

"適切" なセッションを選択するためのこの解は，いかなる意味においても必ず最適な時間割を作り出すようなものではない．最悪な場合には，たとえ同時スケジュールが可能な場合であっても，セッションの数と科目の数が同じになることがあるかも知れない．

1.10　配列，レコードおよび集合構造の表現

プログラミングにおける抽象化の本質的な役割は，その抽象的世界を支配している法則のみにもとづいてプログラムを考え，理解し，そして検証できることであり，特定の計算機の上でその抽象化がどのように実現され，表現されるかということについての洞察や知識が不必要となることである．しかしながら，基本的データ構造のような，プログラムの抽象化に関する基本概念の表現に広く用いられている手法を理解していることは，プログラマとして成功するために有用である．それは，プログラマが，ただ単に構造の抽象的性質だけではなく，計算機の特定な能力や制限を考え，実際の計算機上での実現方法をも考慮してプログラムやデータの設計に関する意味ある決定をすることができるからである．

表 1.3 基本データ型

構造	宣言	選択子	成分へのアクセス	成分型	基数
配列	a: **array**[I] **of** T_0	$a[i]$ $(i \in I)$	計算可能なインデックス i を持った選択子	全て同じ (T_0)	$card(T_0)^{card(I)}$
レコード	r: **record** $s_1 : T_1$; $s_2 : T_2$; … $s_n : T_n$ **end**	$r.s$ $(s \in \{s_1 \ldots s_n\})$	宣言された成分名 s を持った選択子	個々に異なっていてよい	$\prod_{i=1}^{n} card(T_i)$
集合	s: **set of** T_0	なし	関係演算子 in による要素判定	全て同じ (かつ，スカラ型 T_0)	$2^{card(T_0)}$

データ表現の問題は，抽象構造を計算機の記憶装置の上に写像するという問題である．計算機の記憶装置 (*store*) は，第一近似としては，**語** (*word*) と呼ばれる個々の記憶セルからなる配列である．語のインデックスは**番地** (*address*) と呼ばれる．

$$\textbf{var}\ store : \textbf{array}[address]\ \textbf{of}\ word \qquad (1.31)$$

型 *address* および *word* の基数は計算機によって異なっている．*word* の基数が非常に多様であることが特に問題である．その対数は**語の長さ**とよばれる．なぜなら，それは記憶セルを構成しているビットの数だからである．

1.10.1　配列の表現

配列構造の表現とは，成分の型が T の（抽象的）配列を，成分の型が *word* の配列である記憶装置の上に写像することである．

配列は，配列成分の番地計算がなるべく簡単（したがって効率的）であるように写像されなければならない．j 番目の配列成分の番地あるいは記憶インデックス i は，線形関数

$$i=i_0+(j-1)*s \qquad (1.32)$$

によって計算される．ここで i_0 は最初の成分の番地であり，s は1つの成分が“占有”する語の数である．語は，その定義から個別的にアクセスされる記憶の最小単位であるので，s は整数，最も簡単な場合には $s=1$，であることが特に望ましいことは明らかである．もし，s が整数でない場合には(これが普通の場合ではあるが)，s は s を越える最小整数 $\lceil s \rceil$ に切り上げられるのが普通である．

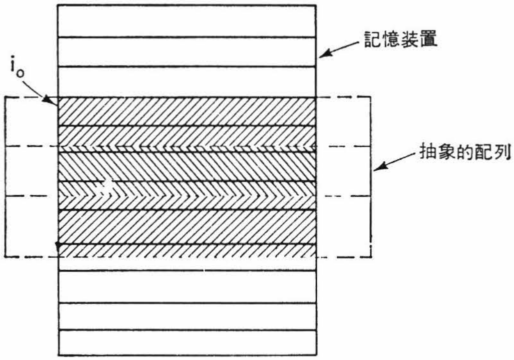


図 1.5　配列の記憶装置への写像

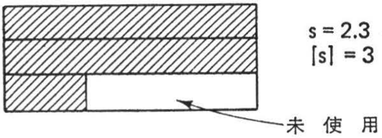


図 1.6 パディングされたレコードの表現

このとき，各配列成分は $\lceil s \rceil$ 語を占めるので，$\lceil s \rceil - s$ 語は使用されないことになる (**図 1.5** および**図 1.6** 参照).
必要語数を次の整数に切り上げることは，**パディング** (padding) と呼ばれる.
記憶装置の利用率 u は，ある構造を表現するのに必要な最小の記憶量を，実際に使った量で割ったものである.

$$u = \frac{s}{s'} = \frac{s}{\lceil s \rceil} \tag{1.33}$$

インプリメントする人は，記憶利用率をなるべく 1 に近づけることを目標としなければならないであろうが，また一方では，語の一部分にアクセスするのは面倒で比較的能率が悪いので妥協が必要となろう．考慮すべき事柄は次の通りである.

1. パディングは記憶利用率を減少させる.
2. パディングの省略には，部分語への非能率なアクセスが必要である.
3. 部分語へのアクセスは，コード（コンパイルされたプログラム）を長くし，したがってパディングの省略によって得られた利益を殺してしまうかも知れない.

実際には，2.，3. の方がより顕著であるのが普通なので，コンパイラは自動的にパディングをするのが常である．利用率は，もし $s>0.5$ なら常に $u>0.5$ となることがわかる．しかしながら，もし $s \leq 0.5$ ならば，1 個以上の配列成分を各語に詰めることによって利用率は著しく増加する．この手法は**パッキング** (packing) と呼ばれる．もし n 個の成分が 1 語に詰められるならば，利用率は次のようになる (**図 1.7** 参照).

$$u = \frac{n \cdot s}{\lceil n \cdot s \rceil} \tag{1.34}$$

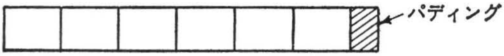


図 1.7 6 成分の 1 語へのパッキング

パックされた配列の i 番目の成分にアクセスするには，その目的の成分を含む語のアドレスの計算と，その語中でのその成分に対応する位置の計算とが必要となる．

$$j = i \ \mathbf{div}\ n$$
$$k = i \ \mathbf{mod}\ n = i - j * n$$

大部分のプログラム言語では，プログラマには抽象的データ構造の実現方法についての権限が与えられていない．しかしながら，少なくとも次のような場合には，プログラマがパッキングの希望を出せるようになっているべきである．それは，1語に1個以上の成分がちょうど（パディングせずに）詰め込め，2倍以上の記憶の経済性が得られる場合である．パッキングの希望を出すための方法として，われわれは宣言中の記号 **array**（または **record**）の前に記号 **packed** をつけることにする．

［例］

$$\mathbf{type}\ alfa = \mathbf{packed\ array}\,[1\,..\,n]\ \mathbf{of}\ char$$

この機能は，語長が長く，語の部分フィールドへのアクセスが比較的便利であるような計算機では特に価値がある．この接頭語の本質的な性質は，これが決してプログラムの意味（あるいは正当性）を変えることがないということである．このことは，プログラムの意味が変わらないという暗黙の保証のもとに，いくつかの表現の間の選択を容易に提示することができることを意味している．

パックされた配列の成分をアクセスするためのコストは，配列全体を一度に解きほぐす（またはパックする）ことにすれば，多くの場合，大幅に減らすことができる．その理由は，配列全体を効率的に順次走査することができ，個々の成分ごとに複雑な写像関数を評価する必要がないからである．したがってわれわれは，次のように定義される2つの標準関数 *pack* と *unpack* の存在を仮定しよう．変数 u, p が

$$u : \mathbf{array}\,[a\,..\,d]\ \mathbf{of}\ T$$
$$p : \mathbf{packed\ array}\,[b\,..\,c]\ \mathbf{of}\ T$$

のようであり，$a \le b \le c \le d$ はすべて同じスカラ型であるとする．このとき，

$$pack(u, i, p), \qquad (a \le i \le b - c + d) \tag{1.36}$$

は

$$p[j]:=u[j+i-b], \quad j=b, \cdots, c$$

に等価であり，そして

$$unpack(p, u, i), \quad (a \leq i \leq b-c+d) \tag{1.37}$$

は，次のものに等価である．

$$u[j+i-b]:=p[j], \quad j=b, \cdots, c$$

1.10.2. レコード構造の表現

レコードは，単にその成分を順に並べることによって計算機の記憶装置上に写像（割付け）できる．レコード r において，その成分（フィールド）r_i の原点番地に対する相対番地 k_i は，その成分の**オフセット**と呼ばれる．それは

$$k_i=s_1+s_2+\cdots+s_{i-1} \tag{1.38}$$

のように計算されるが，ここで，s_j は j 番目の成分のサイズ（語数）である．配列の場合には，その成分がすべて同一の型であるので，

$$s_1=s_2=\cdots=s_n$$

であり，したがって

$$k_i=s_1+\cdots+s_{i-1}=(i-1)\cdot s$$

となる．
レコード構造では，その一般性からこのような簡単な線形関数をオフセット計算に用いることは一般には不可能であり，それゆえに，レコードの成分は固定した識別名によってのみ選択されることが要求される．この制限によって，そのオフセットをコンパイル時に知ることができるという都合のよい結果となるが，これによりレコード中のフィールドへのアクセス効率が増加することは良く知られている．

1つの記憶語のなかに複数のレコード成分を詰め込むことができるときには，パッキングの問題が生じる（図1.8参照）．
われわれは再びパッキングの要求を，宣言において記号 **packed** を **record** の前につけることによって示すことができると仮定しよう．オフセットはコンパイラによって計算されるので，語の内部での成分のオフセットも再びコンパイラによって決定され得る．これは，多くの計算機ではレコードのパッキングは

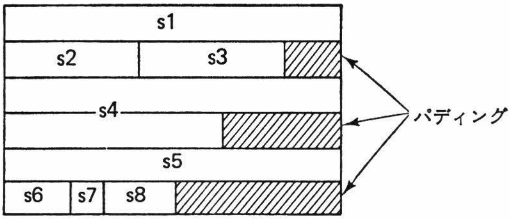


図1.8　パックされたレコードの表現

配列のパッキングに比べてアクセス効率の減少がかなり少ないことを意味している．

1.10.3　集合の表現

計算機の記憶装置上では，集合 s はその**特性関数** $C(s)$ によって表現するのが便利である．これは，論理値からなる配列であり，その i 番目の成分は，その集合中に値 i が存在するか否かを示している．この配列の大きさは集合型の基数によって決定される．

$$C(s_i) \equiv (i \textbf{ in } s) \qquad (1.39)$$

1つの例として，小さな整数からなる集合

$$s = [1,\ 4,\ 8,\ 9]$$

は，その基底型が整数からなる区間型0..9のときには，論理値 F（偽）および T（真）からなる系列によって表現される．

$$C(s) = (FTFFTFFFTT)$$

計算機の記憶装置上では，論理値からなる系列はいわゆるビット列によって表わされる（**図1.9**参照）．

特性関数による集合の表現には，2つの集合の和集合，共通集合および差集合を計算する演算が基本論理演算によって実現できるという利点がある．次の等価関係は，論理演算と集合演算を関係づけたものであり，集合 x および y の基底型に属する任意の要素 i について成立する．

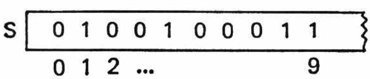


図1.9　集合のビット列による表現

$$
\begin{aligned}
i\,\mathbf{in}\,(x+y) &\equiv (i\,\mathbf{in}\,x)\ \vee\ (i\,\mathbf{in}\,y)\\
i\,\mathbf{in}\,(x*y) &\equiv (i\,\mathbf{in}\,x)\ \wedge\ (i\,\mathbf{in}\,y)\\
i\,\mathbf{in}\,(x-y) &\equiv (i\,\mathbf{in}\,x)\ \wedge\ \neg(i\,\mathbf{in}\,y)
\end{aligned}
\tag{1.40}
$$

論理演算はすべてのディジタル計算機に備わっており，さらにそれらは，語の対応するすべての要素（ビット）に**同時**に作用する．したがって，効率のよい基本集合演算が実現できるためには，基本論理演算だけでなくシフト演算も利用可能であるような，一定数の少ない語で集合を表現しなければならない．このとき，要素判定は1つのシフトと，それに引き続く（符号）ビット・テスト演算によって実現される．その結果，

$$x\,\mathbf{in}\,[c_1,\ c_2,\ \cdots,\ c_n]$$

の形の判定は，それと等価な紋切り型の論理式

$$(x=c_1) \vee (x=c_2) \vee \cdots \vee (x=c_n)$$

よりよほど能率的に実現できる．

結論として，集合構造は**小さな基底型**の場合にのみ用いられるべきである．それなりに能率的な実現が保証されるための基底型の基数の限界は，考えている計算機の語長によって決められる．

この点においては，語長の長い計算機が望ましいことは明白である．もし語長が比較的短いならば，複数語を用いた集合の表現が選ばれるであろう．

1.11　順ファイル構造

今まで述べてきたデータ構造，すなわち配列，レコードおよび集合構造に共通な特徴は，（もしその成分型の基数が有限ならば）その**基数が有限**であることである．したがって，その実現には何ら困難はなく，どんなディジタル計算機上でも簡単に適当な表現が見つけられる．

大部分のいわゆる高度な構造――列，木，グラフ等――は，その基数が無限であることによって特徴づけられる．有限基数をもった基本構造とのこの相違は非常に重要であり，実際的な意味でも重要である．たとえば，**列**構造を次のように定義される：

> 基底型が T_0 の列とは，空列であるか，または型 T_0 の値と基底型が T_0 の列との**連接**である．

このように定義された列型 T は無限の値から成る．それぞれの値が含んでいる型 T_0 の成分の数は有限であるが，この数は有界ではない．すなわち，ど

のような列に対しても，もっと長い列を構成することが可能である．

同様な考察が他のすべての“高度な”データ構造についてもあてはまる．この事実の最も重要な帰結は，高度な構造型に属する値を表現するために必要な記憶量は，コンパイル時にはわからないということである．事実，それはプログラムの実行中に変化するかもしれない．これには動的記憶割付けに関するある機構が必要となるが，これにより，記憶セルは，値が“成長”するときには占有されるが，“縮む”ときには，他での使用のために解放される．

したがって，高度な構造の表現は微妙で困難な問題であり，その解が計算過程の効率と記憶装置の使用の経済性に決定的影響を与えることは明白である．適切な選択は，その構造の上で実行される基本演算とその実行頻度についての知識をもとにしてのみ決定することができる．この情報はどれも言語やコンパイラの設計者には未知であるので，（汎用）言語には高度な構造を取り入れないようにする方がよい．したがって，またプログラマも基本構造だけによって問題が取り扱える場合には，高度な構造の使用は避けるべきである．

高度なデータ構造機能をその潜在的使用法を知らずに用意するというジレンマは，大部分の言語やコンパイラでは，すべての高度な構造は構造をもたない要素や基本構造から構成される，という事実を認識し利用することにより回避されている．この場合，もし成分の動的割付けと動的リンクおよび参照を行なうための機能が備わっていれば，任意の構造はプログラマが陽に指定した操作によって生成することができる．これらの高度な構造を生成し操作するための手法は4章で扱われる．

しかしながら，その基数が無限という点では高度な構造ではあるが，非常に広くかつ頻繁に用いられるので，基本構造にそれを含めることがほとんど必須であるような1つの構造，**列** (sequence) がある．列の抽象的概念を述べるために次の用語と記法を導入する．

1.　$\langle\ \rangle$ は空列を表わす．

2.　$\langle x_0 \rangle$ は1つの成分 x_0 からなる列を表わす．これは1成分列とよばれる．

3.　もし $x=\langle x_1, \cdots, x_m \rangle$ と $y=\langle y_1, \cdots, y_n \rangle$ が列ならば，

$$x \,\&\, y=\langle x_1, \cdots, x_m, y_1, \cdots, y_n \rangle \tag{1.41}$$

は x と y の**連接**である．

4.　もし $x=\langle x_1, \cdots, x_m \rangle$ が空でない列であるならば，

$$first(x) = x_1 \tag{1.42}$$

は x の最初の成分を表わす.

5. もし $x = \langle x_1, \cdots, x_n \rangle$ が空でない列であるならば,

$$rest(x) = \langle x_2, \cdots, x_n \rangle \tag{1.43}$$

は x から最初の成分を除いた列である. したがって, 次の恒等関係が成立する.

$$\langle first(x) \rangle \ \& \ rest(x) \equiv x \tag{1.44}$$

これらの記法の導入は, それらが現在の計算機上で動く実際のプログラム中で使用されるであろう, ということを意味するものではない. 事実, 連接演算はその一般形のままでは用いられないし, また列の取扱いが, 注意深く選ばれかつその使用法に関して一定の規律が保証されるような一組の演算の適用によってのみ行なわれる, ということは本質的である. もちろん, これらの演算は列と連接の抽象的概念から定義される. 列演算を注意深く選ぶことによって, インプリメントする人は, 与えられたどの様な記憶媒体上でも適切で効率的な表現を見つけることができる. このことは, 動的記憶割付けに付随した機構を十分に単純化することができ, プログラマがその詳細を考慮せずに仕事ができることを保証するものである.

基本データ構造として導入された列に対しては, 成分へのアクセスが厳密に順次的な特定の演算のみが許されるが, これを明確にするために, この構造は**順ファイル** (sequential file) または単に**ファイル**と呼ばれる. 配列や集合型の定義をそっくり真似て, ファイル型は形式

$$\boxed{\textbf{type } T = \textbf{file of } T_0} \tag{1.45}$$

によって定義される. これは0個以上の型 T_0 の成分からなる型 T のファイルを表わしている.

[例]

type *text* = **file of** *char*
type *deck* = **file of** *card*

順次的アクセスの本質は, いかなるときでもただちにアクセス可能な成分が特定な1つのものだけであることである. この成分はアクセス機構の**現在位置**に

よって指定される．このファイル位置は，形式的には，ファイルxをそれより左の部分x_Lと右の部分x_Rから構成されるものとして表現する．方程式

$$x \equiv x_L \,\&\, x_R \tag{1.46}$$

が不変な関係を表現していることは明白である．

さて，順次的アクセスの重要な結果として，列の作成と走査のプロセスは別であり，任意の順序に混合することはできない．したがって，ファイルは（その端に）成分を付加することを繰り返すことにより作られ，しかる後，順次的走査により検査されるであろう．ゆえに，ファイルは，作成され（書かれ）ている状態および走査され（読まれ）ている状態という2つの状態のいずれかであると考えるのが普通である．

厳密に順次的なアクセスの利点は，ファイルが2次記憶媒体上に割り付けられ，異なった媒体間での転送が含まれる時には特に顕著である．そのような転送に要する複雑な機構をプログラマから上手に隠すことができるのは，順次的アクセス法だけである．特に，それは簡単な**バッファリング手法**の適用を可能にするが，これによって複雑な計算機システム内の種々の資源の最適な使用を保証することができる．

記憶媒体の中には，順次アクセスしか行なえないものが存在する．この中には全ての種類のテープが含まれる．しかし，磁気ドラムやディスクについても，各記録トラックは順次的アクセスのみが可能である．厳密に順次的なアクセスは，すべての機械的に動く装置や他のいくらかの装置の基本的特徴である．

1.11.1　基本ファイル演算子

次に，プログラマが使用できる具体的ファイル演算子の集合を通して，順次的アクセスの抽象概念を定式化しよう．それらは，列と連接の概念によって定義される．ファイル生成のプロセスを開始する演算子，走査を開始する演算子，列の最後に成分を付加する演算子，および次の成分に走査を進めるための演算子がある．ここで，最後の2つはバッファを表わす暗黙の補助変数を含んだ形式をしていると仮定する．そのようなバッファはすべてのファイル変数xに自動的に関連づけられており，われわれはそれを$x\uparrow$で表わす．明らかに，xが型Tであるとき，$x\uparrow$はその基底型T_0をもつ．

1.　空列の構成．演算

$$rewrite(x) \tag{1.47}$$

は次の代入を表わす．

$$x := \langle\ \rangle$$

この演算は現在の x を書き直し，新しい列を構成するプロセスを開始するために用いられる．すなわちテープの巻き戻しに対応する．

2. 列の延長．演算

$$put(x) \tag{1.48}$$

は，次の代入を表わすが，これは実質的には，列 x に値 $x\uparrow$ を付加する．

$$x := x \mathbin{\&} \langle x\uparrow\rangle$$

3. 走査の開始．演算

$$reset(x) \tag{1.49}$$

は，次の同時代入を表わす．

$$\begin{aligned} x_L &:= \langle\ \rangle \\ x_R &:= x \\ x\uparrow &:= first(x) \end{aligned}$$

この演算は，列を読むプロセスを開始するために用いられる．

4. 次の成分への移行．演算

$$get(x) \tag{1.50}$$

は，次の同時代入を表わす．

$$\begin{aligned} x_L &:= x_L \mathbin{\&} \langle first(x_R)\rangle \\ x_R &:= rest(x_R) \\ x\uparrow &:= first(rest(x_R)) \end{aligned}$$

$first(s)$ は $s \neq \langle\ \rangle$ のときのみ定義されることに注意せよ．

明らかに，演算子 *rewrite* と *reset* は実行前のファイル位置には依存しない．それらは，いかなる場合でもファイルをその先頭に再設定する．

列を走査しているときには，列の終わりを認識できることが必要である．なぜなら，もしそうでないと代入

$$x\uparrow := first(x_R)$$

が未定義演算になってしまうからである．ファイルの終わりに達することは，その右部分 x_R が空になるという条件と同じである．したがって，次の述語

$$eof(x) \equiv x_R = \langle\ \rangle. \tag{1.51}$$

を導入する．これはファイルの終わり (end of file) に達したことを意味している．したがって演算 *get*(*x*) が実行可能なのは，*eof* (*x*) が *false* の場合だけである．

原理的には，ファイルについてのすべての操作をこれら4つの基本ファイル演算子だけで表現することが可能である．しかし実際には，ファイル位置を進める演算 (*get* または *put*) とバッファ変数をアクセスすることを一緒に行なうことが自然な場合が多い．したがって，われわれはさらに2つの手続きを仮定するが，それらは基本演算子で表現できるものである．*v* を変数とし，*e* をファイル成分型 T_0 の式とする．このとき，

$$read(x, v),$$

は

$$v := x\uparrow;\ get(x)$$

と等価であり，また

$$write(x, e).$$

は

$$x\uparrow := e;\ put(x)$$

と等価である．

get と *put* の代わりに *read* と *write* を使う利点は，その簡潔さにあるだけではなく，概念上の単純さにもある．なぜならこのときには，未定義になることがあるようなバッファ変数 $x\uparrow$ の存在を無視することができるからである．しかし，バッファ変数は“先読み”のためには有用であろう．

この2つの手続きの実行に必要な条件は次のようである．

$$\neg eof(x) : read(x, v) \text{ に対して}$$
$$eof(x) : write(x, e) \text{ に対して}$$

読むときには，述語 *eof* (*x*) はファイル *x* の最後の成分が読まれた直後に *true* となる．これらを考慮して，ファイル *x* の順次的構成と処理のための2つの

プログラム図式を次のように構成する．ここで，文 R と S および述語 p は，この図式の付加的パラメータである．

ファイル x への書き込み：

```
rewrite(x);
while p do
    begin R(v); write(x,v)
    end
```
(1.52)

ファイル x からの読み出し：

```
reset(x);
while ¬eof(x) do
    begin read(x,v); S(v)
    end
```
(1.53)

1.11.2 部分構造のあるファイル

大多数のアプリケーションでは，大きなファイルにはある種の部分構造が必要である．たとえば本は1つの文字列と見ることもできるが，章や節に分けられている．部分構造の目的は，長い情報の列の中での位置決めを容易にするために，ある種の明確な参照点や座標を与えることである．実際の記憶装置のなかには，そのような参照点，（たとえばテープ・マーク）を表現する機能をもち，そのような点の間のすべての情報を走査するよりずっと高速に参照点の位置決めを行なうことができるものが多い．

われわれの記法の枠組のなかで，部分構造に関する最初のレベルを導入するための自然な方法は，そのようなファイルを列自身を成分とする列，すなわちファイルのファイルとみなすことである．このとき，最終的な成分（または単位）の型が U であるとすると，部分構造の型は

$$T' = \textbf{file of } U$$

であり，全体のファイルの型は次のようになる．

$$T = \textbf{file of } T'$$

このようにして，任意のネストの深さまで分割されたファイルが構成可能なことは明らかである．一般に，型 T_n を再帰的な関係

$$T_i = \textbf{file of } T_{i-1} \qquad i=1, \cdots, n$$

と $T_0=U$ によって定義することができる．そのようなファイルは，しばしば**多レベル・ファイル**と呼ばれ，型 T_i の成分は，レベル i の**セグメント**といわれる．多レベル・ファイルの例には本があるが，そこでのセグメントのレベルには章，節，項および行が対応する．しかしながら普通は，1レベルのセグメントをもったファイルが断然多いことはもちろんである．

この1レベル・セグメント・ファイルは，決してファイルからなる配列と同じではない．なぜなら，セグメントの数は可変であり，ファイルはその端を延長することが可能だからである．これまで導入してきた記法を使うと，もしあるファイルが

$$x: \textbf{file of file of } U$$

によって定義されているときには，$x\uparrow$ は現在アクセス可能なセグメントを，また，$x\uparrow\uparrow$ は現在アクセス可能な単位成分を表わしている．したがって，*put*$(x\uparrow)$ および *get*$(x\uparrow)$ は単位成分に関するものであり，これに対して *put*(x) および *get*(x) は，セグメントの付加および次のセグメントへの移行を表わす．

セグメント化されたファイルは，実質的にはテープを含むすべての順次的記憶装置上で容易に実現することができる．それらのセグメント化は，個々の成分や（たぶんより高速なスキップ機構によって）セグメントは順次的にのみアクセス可能である，という基本的性質を変えてはいない．他の記憶装置（特に磁気ドラムやディスク）は，多数のトラックをもっており，それらは真に順次的な記憶装置であるが，それだけでは全ファイルを格納するには小さすぎるのが普通である．その結果，ディスク上のファイルは，複数個のトラックに分散され，したがってディスク上にはそれらのトラックの結合に関する適当な管理情報が含まれる．トラックの開始位置は自然なセグメント・マーカになっているが，これが純粋に順次的な媒体上のマーカより直接的かつ容易にアクセス可能なことは明らかである．たとえば，主記憶中のインデックス表を使って，セグメントの始まるトラックや，セグメントの実際の長さを示すことができるであろう（図 1.10 参照）．

これから，いわゆる**インデックスつきファイル**（ときには**直接アクセス・ファイル**とも呼ばれる）が導かれる．実際にはドラムやディスクの各トラックには，読み書きの出発点となる物理的マークが多数含まれている．したがって，各セグメントが全トラックを占める必要はない．なぜなら，もしそのようにす

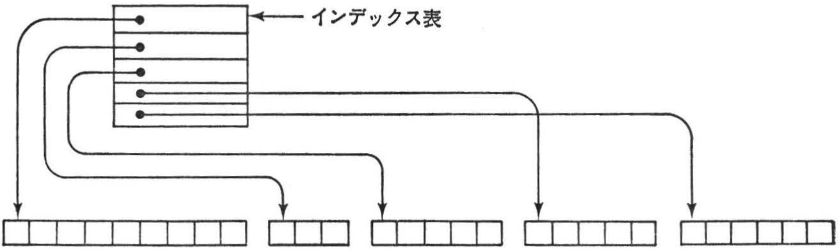


図1.10 5個のセグメントからなるインデックスつきファイル

ると，トラック長に比べてセグメントが短い場合には，記憶利用率が悪くなるからである．2つのマーク間の記憶領域は**物理セグメント**（または**セクタ**）と呼ばれる．これに対し，プログラム中のデータ構造における意味のある実体を**論理セグメント**という．明らかに，どの物理セグメントもたかだか1つの論理セグメントしか収容することができないし，またどの論理セグメントも（空であったとしても）少なくとも1つの物理セグメントを占有する．たとえ“直接アクセス”ファイルと呼ばれているとしても，セグメントの位置ぎめのための平均時間，いわゆる**潜伏**（latency）時間がディスクの回転時間の半分であることは覚えておくべきである．

インデックスつきファイルは，書き込みがその端に順次的に行なわれるという本質的特性を保持している．したがって，それは変更が比較的まれにしか起こらないような応用には特に有用である．変更は，延長かあるいはファイル全体を複写・更新するかのいずれかによって行なわれる．検索はインデックス点を介して選択的かつ高速に行なうことができる．この機能はいわゆる**データ・バンク**において頻繁に利用される．

ファイルの中央部の選択的書き換えを許すようなシステムは，一般には，使用するのがむずかしくかつ危険である．なぜなら，情報中の新しい部分は書き換えられる古い部分と同じ大きさでなければならないからである．そのうえ多量のデータを含むようなアプリケーションでは，障害——それがプログラムの誤りによるものであろうと，機械の誤動作によるものであろうと——が発生したときその失敗したプロセスを再実行するためのデータの状態が存在すべきである，という原則からも選択的更新は推奨されない．更新の目的からは，順編成は信頼性の点で最も優れた方法である．この方法は，多量のデータ集合についてのより技巧的な編成——より効率的であるが，障害時にはデータが完全に

失われてしまうかもしれないような――より多用されるべきである†.

1.11.3 テキスト

その成分型が *char* であるようなファイルは，計算やデータ処理では特に重要な役割を演じる．なぜなら，それは計算機とユーザである人間との間のインターフェイスであるからである．計算結果を表わす“読める出力”と同様に，プログラマが与える“読める入力”は文字からなる列である．それゆえ，このデータ型には標準名

type *text*=**file of** *char*

を与える．

計算プロセスとその作成者である人間との通信は，究極的には，2つのテキスト・ファイルにより表現されるインターフェイスによって行なわれる．そのうちの1つは計算プロセスに対する**入力** (input) を，またもう1つは**出力** (output) と呼ばれる計算結果を含んでいる．今後は，すべてのプログラム中に次のように宣言された2つのファイルがあると仮定する．

var *input*, *output*: *text*

これらのファイルが，計算機システムの標準入出力装置を表わしているという仮定を考慮して，ファイル *input* は読み出しのみ可能，ファイル *output* は書き込みのみ可能とする．

この2つのファイルは特によく使われるので，手続き *read* と *write* の第1パラメータがファイル変数でないときには，*input* と *output* が省略されたものと考える．さらに，この2つの標準手続きに対しては，任意個のパラメータを書いてもよいとする．これらの記法上の約束は次のように要約される：

read($x1$, ⋯, xn) は *read*(*input*, $x1$, ⋯, xn) を，
write($x1$, ⋯, xn) は *write*(*output*, $x1$, ⋯, xn) を，
read(f, $x1$, ⋯, xn) は
　begin *read* (f, $x1$)；⋯；*read*(f, xn) **end** を，そして
write (f, $x1$, ⋯, xn) は
　begin *write*(f, $x1$)；⋯；*write*(f, xn) **end** を

それぞれ表わす．

---

† **訳注**　全体的に本書でのファイルの取扱いは，現在のOSの提供するデータ管理機能から見ると少し控え目すぎるように思われる．

テキストは部分構造をもつ列の典形的な例である．部分構造の通常の単位は章，節，および行である．テキスト・ファイル中でこの部分構造を表現するためによく用いられる方法は，特殊な区切り記号を用いることである．空白文字はこの最も良く知られた例であるが，同様な区切り記号が，行，節や章の終わりに使われるかも知れない．たとえば広く使われている ISO 文字集合（そのアメリカ版 ACSII も含めて）には，**制御文字**と呼ばれるいくつかのそのような要素が含まれている（補遺 I 参照）．

本書では，特殊な区切り記号による特定な方法で部分構造の表現を行なうことは避ける．その代わり，われわれは，各行を表わす文字列からなるファイルとしてテキストを考える．また，部分構造のレベルとしては，1 レベルすなわち行のみを議論の対象とする．しかしながら，テキストを印刷可能文字からなるファイルのファイルとは見ずに，単に文字からなるファイルと考え，行を操作する，すなわち，印をつけ認識をするための付加的な演算子や述語を導入する．これらの演算子や述語の意味は，行が（型 *char* に属さない仮想的な）区切り記号によって区切られており，そのような区切り記号の挿入や認識がその役割であると考えれば最も良く理解できるであろう．付加的な演算子は次のようである．

writeln(*f*)	ファイル *f* に行マークを付加する
readln(*f*)	ファイル *f* において，次の行マークの直後の文字まで読みとばす
eoln(*f*)	ファイル位置が行マークに達したとき *true*，そうでないとき *false* となる論理関数 [*eoln* (*f*) が *true* のとき，$f\uparrow$=空白，と仮定する]

次に，これらを用いてテキストに対する読み出しと書き込みのためのプログラム図式を定式化するが，これらは他のファイルに対する“読み出し”と“書き込み”のものと同様である［(1.52) と (1.53) 参照］．これらの図式では，テキスト・ファイル *f* を仮定し，また行構造の生成と認識には相応な注意が払われている．$R(x)$ は（型 *char* の）x に値を代入する文とし，条件 p および q は，“これは行の最後の文字であった” および “これはファイルの最後の文字であった” と定義する．U は各行を読んだ直後に実行される文とし，$S(x)$ はファイルの各文字 x について実行される文，そして，V は各行の終わりで実行される文とする．

テキスト f への書き込み：

```
rewrite(f);
while ¬q do
    begin
        while ¬p do
            begin R(x); write(f,x)
            end ;
        writeln(f)
    end
```
(1.54)

テキスト f からの読み出し：

```
reset(f);
while ¬eof(f) do
    begin U;
        while ¬eoln(f) do
            begin read(f,x); S(x)
            end ;
        V; readln(f)
    end
```
(1.55)

行構造が特に意味のある情報をもっていないという場合がある．このような場合には，行マークに出合ったときのバッファ変数についてのわれわれの仮定[*eoln*(f)の定義参照]から，プログラム図式が簡単になる．*eoln* の定義に従えば，行の終わりは付加的な空白記号に見えることに注意してほしい．

```
while ¬eof(f) do
    begin read(f,x); S(x)
    end
```
(1.56)

大部分のプログラム言語では，型"整数"や"実数"の引数を入出力手続きに許しているのが普通である．このような一般化は，型"整数"や"実数"が，その数の各桁を表わす成分からなる文字配列として定義されていれば簡単であろう．実際，商業上の応用を特に考慮した言語ではこのような定義を守っており，10進桁や10進数システムでの数の表現を用いている．データ型**整数**や**実**

数を基本型として導入した重要な利点は，そのような詳細な仕様を省略することができ，かつ計算システムではその目的により適している異なった数表現を使用することができることである．事実，科学計算用システムでは，常に2進表現を用いてるが，これはほとんどの点で10進表現より優れている．

しかしながら，このとき，テキスト・ファイルから(へ)の入(出)力には変換操作が不可欠であることをプログラマは誰でも知っているべきである．このような変換操作は，数型の引数をもった入出力文の陰に隠されるのが普通である．しかし，職業的なプログラマなら，そのような文（いわゆるI/O文）には2つの機能，異なった媒体間のデータ転送とデータ表現の変換，が組み込まれていることを知っている．後者は非常に複雑で時間のかかる仕事である．

本書のこれ以後の章では，プログラム言語PASCALでの規則に従って数を引数とする入出力文を使うことにする．この規則には，変換過程を制御するためにある種のフォーマット指定が許されている．出力文の場合には，このフォーマット指定で希望桁数を示す．この文字数はまた“フィールド幅”とも呼ばれ，引数の直後に次のように書く．

$$write(f, x: n)$$

引数 x がファイル f 上に書かれるが，その値は（少なくとも）n 個の文字からなる列に変換され，もし必要なら，符号または適当な数の空白文字が数字列の前に置かれる．

本書のプログラム例を理解するには，これ以上の詳細は不必要である．ここでは，数表現の変換のための2つのプログラム例（**プログラム1.3と1.4**）を与えるが，これは，通常は出力文の中で暗に仮定されているそのような操作が非常に複雑であることを示すためである．これらの2つの手続きは，実数の10進表現から任意の内部表現への，あるいはその逆の変換を表わしている．(初めの部分の諸定数は，CDC 6600 計算機の浮動小数点形式の性質から決められたものである：11ビットの2進指数部と48ビットの仮数部．関数 *expo* (x) は x の指数部を表わす)．

プログラム1.3　実数の入力

```
procedure readreal (var f: text; var x: real);
   { read real number x from file f}
   {the following are system dependent constants}
   const t48 = 281474976710656;      {= 2**48}
      limit  =  56294995342131;      {= t48 div 5}
      z = 27;      {= ord('0') }
      lim 1 = 322;      { maximum exponent }
      lim 2 = -292;   { minimum exponent }
   type posint = 0 .. 323;
   var ch: char; y: real; a,i,e: integer;
      s,ss: boolean;      { signs }
   function ten(e: posint): real; { = 10**e, 0<e<322 }
      var i: integer; t: real;
   begin i := 0; t := 1.0;
      repeat if odd(e) then
         case i of
          0: t := t * 1.0E1;
          1: t := t * 1.0E2;
          2: t := t * 1.0E4;
          3: t := t * 1.0E8;
          4: t := t * 1.0E16;.
          5: t := t * 1.0E32;
          6: t := t * 1.0E64;
          7: t := t * 1.0E128;
          8: t := t * 1.0E256
         end ;
         e := e div 2; i := i+1
      until e = 0;
      ten := t
   end ;
begin
   {skip leading blanks}
   while f↑=' ' do get(f);
   ch := f↑ ;
```

```
if ch = '−' then
   begin s := true; get(f); ch := f↑
   end else
   begin s := false;
      if ch = '+' then
      begin get(f); ch := f↑
      end
   end ;
if ¬(ch in ['0' .. '9']) then
begin message (' DIGIT EXPECTED'); halt;
end ;
a := 0; e := 0;
repeat if a < limit then a := 10*a + ord(ch)−z else e := e+1;
   get(f); ch := f↑
until ¬(ch in ['0' .. '9']);
if ch = '.' then
begin { read fraction } get(f); ch := f↑;
   while ch in ['0' .. '9'] do
   begin if a < limit then
         begin a := 10*a + ord(ch)−z; e := e−1
         end ;
      get(f); ch := f↑
   end
end ;
if ch = 'E' then
begin { read scale factor } get(f); ch := f↑;
   i := 0;
   if ch = '−' then
   begin ss := true; get(f); ch := f↑
   end else
   begin ss := false; if ch = '+' then
      begin get(f); ch := f↑
      end
   end ;
   while ch in ['0' .. '9'] do
   begin if i<limit then begin i := 10*i + ord(ch)−z end;
      get(f); ch := f↑
   end ;
   if ss then e := e−i else e := e+i
end ;
```

```
        if e < lim 2 then
            begin a := 0; e := 0
            end else
        if e > lim 1 then
        begin message(' NUMBER TOO LARGE '); halt end;
        { 0 < a < 2**49 }
        if a ≥ t48 then y := ((a+1) div 2) * 2.0 else y := a;
        if s then y := -y;
        if e < 0 then x := y/ten(-e) else
        if e ≠ 0 then x := y*ten(e) else x := y ;
        while (f↑ = ' ')∧(¬eof(f)) do get(f);
end {readreal}
```

プログラム 1.4　実数の出力

```
procedure writereal (var f: text; x: real; n: integer);
    {write real number x with n characters in decimal flt.pt. format}
    {the following constants depend on the underlying floating-point representa-
    tion of real numbers}
    const t48 = 281474976710656; {= 2**48; 48 = size of mantissa}
          z = 27;      { ord('0') }
    type posint = 0 .. 323;      {range of decimal exponent}
    var c,d,e,e0,e1,e2,i: integer;

    function ten(e: posint): real;      { 10**e, 0<e<322 }
        var i: integer; t: real;
    begin i := 0; t := 1.0;
        repeat if odd(e) then
            case i of
             0: t := t * 1.0E1;
             1: t := t * 1.0E2;
             2: t := t * 1.0E4;
             3: t := t * 1.0E8;
             4: t := t * 1.0E16;
             5: t := t * 1.0E32;
             6: t := t * 1.0E64;
             7: t := t * 1.0E128;
             8: t := t * 1.0E256
            end ;
```

```
            e := e div 2; i := i+1
        until e = 0;
        ten := t
    end { ten } ;
begin { at least 10 characters needed: b+9.9E+999 }
    if x = 0 then
    begin repeat write(f, ' '); n := n-1
            until n ≤ 1;
        write(f, '0')
    end else
    begin
        if n ≤ 10 then n := 3 else n := n-7;
        repeat write(f, ' '); n := n-1
        until n ≤ 15;
        { 1 < n ≤ 15,  number of digits to be printed }
        begin { test sign, then obtain exponent }
            if x < 0 then
                begin write(f, '-'); x := -x
                end else write (f, ' ');
            e := expo (x);  {e = entier(log2(abs(x)))}
            if e ≥ 0 then
                begin e := e*77 div 256 +1; x := x/ten(e);
                    if x ≥ 1.0 then
                        begin x := x/10.0; e := e+1
                        end
                end else
                begin e := (e+1)*77 div 256; x := ten(-e)*x;
                    if x < 0.1 then
                        begin x := 10.0*x; e := e-1
                        end
                end ;
            { 0.1 ≤ x < 1.0 }
            case n of      { rounding }
              2: x := x+0.5E-2;
              3: x := x+0.5E-3;
              4: x := x+0.5E-4;
              5: x := x+0.5E-5;
              6: x := x+0.5E-6;
              7: x := x+0.5E-7;
```

```
         8: x := x+0.5E-8;
         9: x := x+0.5E-9;
        10: x := x+0.5E-10;
        11: x := x+0.5E-11;
        12: x := x+0.5E-12;
        13: x := x+0.5E-13;
        14: x := x+0.5E-14;
        15: x := x+0.5E-15
      end ;
      if x ≥ 1.0 then
         begin x := x * 0.1; e := e+1;
         end ;
      c := trunc(x,48); {= trunc(x*(2**48))}
      c := 10*c; d := c div t48;
      write(f, chr(d+z), '.');
      for i := 2 to n do
      begin c := (c - d*t48) * 10; d := c div t48;
         write(f, chr(d+z))
      end ;
      write(f, 'E'); e := e-1;
      if e < 0 then
         begin write(f, '-'); e := -e
         end else write(f, '+');
      e1 := e * 205 div 2048; e2 := e - 10*e1;
      e0 := e1 * 205 div 2048; e1 := e1 - 10*e0;
      write(f, chr(e0+z), chr(e1+z), chr(e2+z))
    end
  end
end {writereal}
```

1.11.4　ファイル編集プログラム

順次的構造の応用例として次の問題を提出するが，これはプログラムの作成とその説明を行なうための方法を示すのにも役立つ．この方法は，**段階的細詳化** (stepwise refinement) [1-4, 1-6] と呼ばれ，本書を通して多くのアルゴリズムの説明に使われるであろう．

この問題は，テキスト x を編集して y を作るプログラムを作成することである．ここで編集とは，特定の行の削除・取換えおよび新しい行の挿入を意味

する．編集は，次のような**編集命令**の列によって制御されるが，この列は標準ファイル *input* によって表わされる．

I, *m*.	*m* 番目の行の後にテキストを挿入する
D, *m*, *n*.	行 *m* から *n* まで削除する
R, *m*, *n*.	行 *m* から *n* まで取り換える．
E.	編集プロセスを停止する

各命令は，命令ファイルと呼ばれる標準ファイル *input* の1行として書かれている．*m* と *n* は10進数の行番号である．挿入すべきテキストは，I および R 命令の直後に書き，空行によってその終わりを示す．

編集命令は，それに含まれる行番号が厳密に増加する順番に現われるように出されると仮定する．この規則によって，入力テキスト x は厳密に**順次的に処理**できることになる．明らかに，処理の状態は，現在調査中の行の番号によって示される x の現在位置によって表わすことができる．

さて，この編集プログラムは対話的に使われるとし，したがって命令ファイルは，たとえば，キー・ボードから入力されるデータを表わしていると仮定する．このような操作の形態では，オペレータがある種のフィードバックを受けることが非常に望ましい．適切かつ有用なフィードバックのひとつに，最後の命令によって処理された次の行のテキストがある．この行を**現在行**と呼ぶ．各命令の実行後にこの現在行を印刷するという新しい要求を満たすためには，現在行をある変数によって陽に表現し，行を x から読んだ後とそれを y に書き出す前に，その行をこの変数に貯えなければならない．
この手法は“先読み”と呼ばれる．編集プログラムは次のように構成することができる．

```
program editor (x, y, input, output);
var lno: integer;       {number of current line}
    cl : line;          {current line}
    x,y: text;
begin read instruction;
   repeat interpret instruction;                        (1.57)
          write line;
          read instruction
   until instruction = 'E'
end.
```

次に，各文をもっと詳細に規定してゆこう．*read instruction*（命令を読み込む）と *interpret instruction*（命令を解釈する）の詳細化については，一般に，1つの命令（instruction）は3つの部分，すなわち命令コードと2つのパラメータから成っていることに注意しよう．したがって，この2つのルーチンとの通信のために3つの変数 *code*, *m*, *n* を導入する．

```
var code,ch: char;
    m,n: integer
```

***read instruction*:**

```
read(code,ch);
if ch = ',' then read(m,ch) else m := lno;
if ch = ',' then read(n) else n := m
```
(1.58)

この構成は，“存在しない”細目については適当な規定値を代入することにより，パラメータが0,1あるいは2個の命令のいずれをも受けつけるようにできている．

***interpret instruction*:**

```
copy;
if code = 'I' then
begin putline;
      insert
end else
if code = 'D' then skip else
if code = 'R' then
begin insert;
      skip
end else
if code = 'E' then copyrest else Error
```
(1.59)

詳細化の2番目のステップとして，(1.59) で用いられた文 *copy*, *insert* および *skip* を，1つの行だけに関する操作を用いて，すなわち *getline* と *putline* を用いて表わす．それらに共通な特徴は繰返し構造である．*copy* は，現在行から始めて m 番目の行で終わるまで，x から y に行を複写する．*skip* は，複写せずに n 番目の行まで x から行を読む．

```
copy:     while lno < m do
          begin putline;
                getline
          end
skip:     while lno < n do getline                              (1.60)
insert:   readline;
          while noend do
              begin putline; readline
              end;
          getline
copyrest: while ¬eof(x) do
              begin putline; getline
              end;
          putline
```

詳細化の3番目の，そして最後のステップは，操作 *getline*, *putline*, *readline*, および *writeline* を1文字についての操作によって表現することである．これまでのすべての操作は行全体についてのものであり，行の詳細な部分構造については，特別な仮定をしていなかったことに注意しよう．行はそれ自身文字からなる列であるので，（現在行を保持している）変数 *cl* は列として宣言したいと思うかも知れない．

var *cl*: **file of** *char*

しかしながら，基本構造（配列のような）で十分なときには基数が無限大であるような構造は使うべきでない，という忠告を思い出して欲しい．もし，1行の長さを，たとえば，80字に制限すればこれは可能であり，したがって次のように指定する．

var *cl*: **array**[1..80]**of** *char*

4つのルーチンは，この配列に関するひとつのインデックス変数 i を用いているが，これは実際には局所的に用いられているので，各ルーチンに対して局所的であると宣言してもよい．さらに，現在行の長さを表わすために，ひとつの大域的変数 L を導入する必要がある．

```
getline:   i := 0; lno := lno + 1;
           while ¬eoln(x) do
               begin i := i+1; read (x, cl[i])
               end ;
           L := i; readln(x)

putline:   i := 0;
           while i < L do
               begin i := i+1; write (y, cl[i])
               end ;
           writeln(y)
readline:  i := 0;                                        (1.61)
           while ¬eoln(input) do
               begin i := i+1; read(cl[i])
               end ;
           L := i; readln
writeline: i := 0; write (lno);
           while i < L do
               begin i := i+1; write(cl[i])
               end ;
           writeln
```

ルーチン *insert* 中の条件 *noend* は，今では次のように簡単に表わすことができる．

$$L \neq 0$$

これでファイル編集プログラムの作成が終了した．

演 習 問 題

1.1　標準型 *integer*, *real* および *char* の基数を C_I, C_R および C_C で表わすとき，本章の例題として定義された次のデータ型の基数はどうなるか：*sex*, *Boolean*, *weekday*, *letter*, *digit*, *officer*, *row*, *alfa*, *complex*, *date*, *person*, *Coordinate*, *charset*, *tapestatus*

1.2　演習問題 1.1 で挙げた型の変数は，次の場合についてどのように表現されるか．

(a)　諸君の計算機の記憶装置

(b) FORTRAN

(c) 諸君の好きなプログラム言語

1.3 (諸君の計算機上で) 次に対応する命令列はどのようになるか.

(a) パックされたレコードと配列に対する成分の取り出しと格納

(b) 要素判定を含んだ集合演算

1.4 可変レコードの正しい使い方は実行時にチェックできるか. さらに, それはコンパイル時にも検証できるか.

1.5 ある種のデータ集合を, 配列としてではなく順ファイルとして定義する理由は何か.

1.6 1.11 節で定義された順ファイルを非常に大きな主記憶をもった計算機上で実現すると仮定しよう. ファイルはある一定長 L より長くならないという制限を加えれば, 配列を用いて表現できる.

データ表現の選択やファイル演算子 *get*, *put*, *reset* および *rewrite* (これらは 1.11 節で 1 組の公理によって定義されている) のための手続きを含めて, 可能な実現法について述べよ.

1.7 セグメント化されたファイルについて, 演習問題 1.6 を考えよ.

1.8 ある鉄道システムの種々の路線の毎日のサービスを載せた鉄道時刻表がある. ある駅と列車の方向が与えられたとき, 到着および出発時刻を調べるのに都合が良いように, 配列やレコードあるいはファイルを用いてこれらのデータを表現する方法を考えよ.

1.9 ファイルの形でテキスト T が, また 2 つの配列 A, B の形で少数の語からなるリストが与えられている. 各語は短かく, 一定の最大長をもった文字配列であるとする.

テキスト T 中の各語 A_i の出現を対応する語 B_i で置き換えて, テキスト S に変換するためのプログラムを書け.

1.10 プログラム 1.3 と 1.4 を諸君の計算機に合わせるためにはどのような調整 (定数の再定義等) が必要か.

1.11 プログラム 1.4 と同様でかつその先頭の文が次のような手続きを書け.

procedure *writereal* (**var** *f*: *text*; *x*: *real*; *n*, *m*; *integer*);

これは, 値 x を固定小数点形式で小数点以下 m 桁まで求め, 少なくとも

n 桁をファイル f に追加するものである．必要ならば，適当な数の空白文字や符号をその前につけよ．

1.12　1.11.4 節のテキスト編集プログラムを完全なプログラムの形に書き直せ．

1.13　2分探索についての次の3つのプログラムを (1.17) と比較せよ．この3つのうちどれが正しいか．どれが最も効率的か．次の変数と $N>0$ を仮定する．

```
var i,j,k: integer;
    a: array[1 .. N] of T;
    x: T
```

プログラム A:

```
i := 1; j := N;
repeat k := (i+j) div 2;
    if a[k] < x then i := k else j := k
until (a[k]=x) ∨ (i≥j)
```

プログラム B:

```
i := 1; j := N;
repeat k := (i+j) div 2;
    if x ≤ a[k] then j := k−1;
    if a[k] ≤ x then i := k+1
until i > j
```

プログラム C:

```
i := 1; j := N;
repeat k := (i+j) div 2;
    if x < a[k] then j := k else i := k+1
until i ≥ j
```

ヒント：すべてのプログラムは，要素 x が存在すれば $a[k]=x$ として停止し，値 x をもつ要素が存在しなければ $a[k]\neq x$ として停止しなければならない．

1.14　ある会社でその製品の成功を判定するために投票を行なった．その製品とはヒット曲のレコードやテープで，最も人気のあるヒット曲は，ヒット・パレードで放送されることになっている．投票を行なう人達はその性別と年齢（たとえば20歳以下と20歳以上）によって，4つのカテゴリに

分けられている．各人は5つのヒット曲を指名する．ヒット曲は，1からN(たとえば$N=30$)までの数によって識別される．投票(*poll*)結果はあるファイルによって表現される．

```
type hit = 1 .. N;
    sex = (male, female);
    response =
        record name, firstname: alfa;
               s: sex;
               age: integer;
               choice: array [1 .. 5] of hit
        end ;
var poll: file of response
```

各ファイル要素は投票者を表わし，彼の名前，洗礼名，性別，年齢，および彼の好きな5曲（好きな順に）をリストしたものである．このファイルから次の結果を計算するプログラムを作れ．

1．人気の順に並んだヒット曲のリスト．各エントリは，ヒット曲とその得票数からなる．投票されなかった曲は，このリストからはずす．
2．各カテゴリごとに，そのカテゴリの中で最も人気のあった3曲のうちどれかを第1位に指名した人の名前と洗礼名からなる4つのリスト．

これら5つのリストには，適当な表題がつけられるとする．

参考文献

1-1. DAHL, O. J., DIJKSTRA, E. W., and HOARE, C. A. R., *Structured Programming*, (New York: Academic Press, 1972), pp. 155–65.

1-2. HOARE, C. A. R., "Notes on Data Structuring," in *Structured Programming*, Dahl, Dijkstra, and Hoare, pp. 83–174.

1-3. JENSEN, K. and WIRTH, N., "PASCAL, User Manual and Report," *Lecture Notes in Computer Science*, Vol. 18 (Berlin: Springer-Verlag, 1974).

1-4. WIRTH, N., "Program Development by Stepwise Refinement," *Comm. ACM*, **14**, No. 4 (1971), 221–27.

1-5. ________, "The Programming Language PASCAL," *Acta Informatica*, **1**, No. 1 (1971), 35–63.

1-6. ________, "On the Composition of Well-Structured Programs," *Computing Surveys*, **6**, No. 4, (1974) 247–59.

第2章

ソーティング

2.1 はじめに

この章の主な目的は，前章で導入されたデータ構造の使い方を示す広範な例題を与え，そして，基本となるデータ構造の選択が，与えられた仕事を行なうためのアルゴリズムにいかに深く影響をおよぼすかを示すことである．**ソーティング** (sorting; 分類，整列) は，また，ひとつの仕事が多くの異なったアルゴリズムによって実行でき，かつ特定の応用という立場から見たときに，それらが他のものに比べて長所や短所をあわせもつことを示すよい実例でもある．

ソーティングは，一般には，与えられた物の集まりをある特定の**順序**に並べ直すプロセスであると理解されている．ソーティングの目的は，**ソート** (sort) された集合の要素に対する後の探索を容易ならしめることである．したがって，それはほとんど普遍的に実行される基本的な活動である．電話帳でも，所得税ファイルでも，目次でも，図書館でも，辞書でも，倉庫でも，そして貯えられているものを探索し取り出すようなすべての場所では，物はソートされている．小さな子供達でも彼らの物を整理するように教えられ，そして彼等が算術を学ぶよりずっと前から，ある種のソーティングに直面している．

したがって，特にデータ処理では，ソーティングは適切かつ本質的な作業である．"データ"よりソートしやすいものがほかにあろうか．それにもかかわらず，ソーティングにおけるわれわれの主な興味は，アルゴリズムの構成に使われるより基本的な手法に向けられている．ソーティング・アルゴリズムとの関係でどこにも現われないような手法は多くはない．特にソーティングは，皆同じ目的をもち，多くのものはある意味で最適であり，そして大部分のものは他より優れた点をもっている，そういった多数のアルゴリズムを示す理想的な題材である．それゆえに，それはアルゴリズムの性能解析の必要性を示す理想的

な題目でもある．さらに，ソーティングの例題は，明らかな方法がただちに使えるような場合でも，凝ったアルゴリズムを開発することにより，非常に大きな性能上の利益が得られることを示すのにも適している．

アルゴリズムの選択が処理すべきデータの構造に依存することは普遍的な現象ではあるが，ソーティングの場合には特に顕著であり，ソーティング法は一般には，**配列のソーティング**と(順)**ファイルのソーティング**という2つのカテゴリに分けられる．これらは，しばしば**内部**および**外部**ソーティングとよばれる．この理由は配列が高速なランダム・アクセス“内部”記憶装置に格納されるのに反し，ファイルは，機械的動作を行なう装置（ディスクやテープ）上の低速ではあるが記憶容量の大きな“外部”記憶に置くのが便利だからである．この違いの重要性は，番号のついたカードをソートするという例題を考えれば明らかである．カードの集合を配列として構成することは，各カードが見えてかつ個々にアクセス可能なようにカードを置くことに相当する(**図2.1**参照)．

これに対し，カードの集合をファイルとして構成することは，各山の最上のカードのみが見えることを意味する（**図2.2**参照)．このような制限は，明らかに使用するソーティング法に重大な結果をもたらすが，もしカードの枚数がテーブルに置けないほど多いときには避けられないことである．

先に進む前にこの章を通して使われる用語や記法をいくつか導入する．項目



図2.1　配列のソーティング



図 2.2　ファイルのソーティング

の集合

$$a_1,\ a_2,\ \cdots,\ a_n$$

が与えられたとき，ソーティングとはこれらの項目をある順序

$$a_{k1}, a_{k2},\ \cdots,\ a_{kn}$$

に並び換えることである．ただし，順序づけ関数 f は次を満たすとする．

$$f\ (a_{k1}) \leq f\ (a_{k2}) \leq\ \cdots\ \leq f\ (a_{kn}) \qquad (2.1)$$

通常，順序づけ関数は，ある特定な計算規則に従って評価されるというのではなく，各項目の特定な成分（フィールド）に格納されている．この値は項目の**キー** (key) と呼ばれる．したがって，項目 a_i を表現するには，レコード構造が特に適している．これ以後のすべてのソーティング・アルゴリズムで使うために，型 *item* を次のように定義する．

```
type item = record key: integer;
                {other components declared here}
            end
```
(2.2)

"*other components*"（他の成分）は，この集合中の項目についての適当なデ

ータを表わしているが，その項目を識別するにはキーだけで十分である．われわれのソーティング・アルゴリズムに関する限り，意味のある成分はキーのみであり，したがって残りの成分を定義する必要はない．キーの型としては，*integer* を選ばなければならないというわけではない．完全な順序関係が定義された型ならどれでもよい．

ソーティング方法は，等しいキーをもった項目間の相対的順序がソーティングによって変化しないとき，**安定**であるといわれる．ソーティングの安定性は，項目がある2次的なキー［(1次的な）キーには反映されない性質］によって既にソートされている場合にはしばしば要求される．

この章は，ソーティング手法についての総括的サーベイではない．むしろ，いくつかの選ばれた特定の方法をより詳細に述べたものである．ソーティングについての全体的な取り扱いに関して興味ある読者は D. E. Knuth [2-7] による優れた包括的概説書を参考にして欲しい（Lorin [2-10] も参照のこと）．

2.2 配列のソーティング

配列のソーティング法に対する第1の要求は，利用可能な記憶領域を経済的に使用することである．このことから，項目を並びかえるための置換は**その場所**で行なわれなければならず，また配列 a を結果の配列 b に転送する方法には本質的にあまり興味がない．多数の解法のなかから記憶の経済性に関するこの評価基準を満たすものをまず考え，次に能率すなわち時間の経済性によって類別を行なう．**能率の測度**としては，必要となる**キーの比較**回数 C や，**項目の移動**（置換）回数 M が適当である．これらの数は，ソートすべき項目の数 n の関数である．良いソーティング・アルゴリズムでは，$n \log n$ のオーダの比較しか使用しないが，われわれは，まず n^2 回のキー比較を必要とするが，簡単で明らかな単純法と呼ばれるいくつかのソーティング方法について議論する．より速いアルゴリズムに先立って単純法を述べるには3つの正当な理由がある：

1. ソーティングの原理の特性を明確にするには，単純法は特に適している．
2. そのプログラムは理解しやすく，短い．プログラムも記憶領域を必要とすることを思い出してほしい
3. 凝った方法はより少ない操作しか必要としないが，それらの操作はその細部がより複雑なのが普通である．その結果として，単純法は，大きな n に対して使うべきではないが，十分小さな n に対してはより高速であ

る．

その場所で項目をソートするソーティング・アルゴリズムは，その基本原理によって次の3つのカテゴリに類別される：

1. 挿入によるソーティング
2. 選択によるソーティング
3. 交換によるソーティング

以下では，これら3つの基本原理について考察し比較を行なう．プログラムは，成分がその場所でソートされる変数 a や，データ型 *item* (2.2) および *index* を操作するが，それらは次のように定義されている：

$$\begin{array}{l} \textbf{type}\ index = 0\,..\,n; \\ \textbf{var}\ a\text{: }\textbf{array}\ [1\,..\,n]\ \textbf{of}\ item \end{array} \qquad (2.3)$$

2.2.1　単純挿入によるソーティング

この方法は，カード・プレイヤによって広く用いられている方法である．項目（カード）の集合は概念的には，目的列 a_1, $\cdots$, a_{i-1} と原列 a_i, $\cdots$, a_n に分けられる．$i=2$ から始めて i を1つずつ増やしながら，各ステップでは原列の i 番目の要素を取り出し，それを目的列の適当な場所に**挿入**する．

ランダムに選ばれた8個の数の例題について，挿入によるソーティングの過程を**表 2.1** に示す．単純挿入法のアルゴリズムは次のようである．

表 2.1　単純挿入によるソーティング過程の例

初期キー	44	55	12	42	94	18	06	67
$i=2$	44	55	12	42	94	18	06	67
$i=3$	12	44	55	42	94	18	06	67
$i=4$	12	42	44	55	94	18	06	67
$i=5$	12	42	44	55	94	18	06	67
$i=6$	12	18	42	44	55	94	06	67
$i=7$	06	12	18	42	44	55	94	67
$i=8$	06	12	18	42	44	55	67	94

```
for i := 2 to n do
  begin x := a_i
    "a_1 ··· a_{i-1} 中の適当な場所に x を挿入する"
  end
```

適当な場所を実際に見つける過程では，比較と移動を交互に行なうのが便利である．すなわち，x と次の項目 a_j とを比較し，x を挿入するか，または a_j を右に動かし x を左に進める——x を"シフト・ダウン"させる．シフト・ダウンの過程を停止させる条件には2つの異なるものがあることに注意しよう．

1．x のキーよりも小さいキーをもった項目 a_j が見つかったとき
2．目的列の左端に達したとき

2つの停止条件をもったこの典形的な繰返しの実行には，いわゆる**番人法**が使える．この場合には，項目 $a_0=x$ を番人として置いておけばよい．(このためには，a の宣言において，インデックス範囲を $0..n$ に拡張しておかなければならない)．完成したアルゴリズムは，**プログラム 2.1** で与えられる．

単純挿入法の解析　i 番目のシフトにおけるキー比較の回数 C_i は，最大 $i-1$，最小 1，また——n 個のキーのすべての置換が等確率で起こるならば——平均 $i/2$ である．移動（項目の代入）の回数 M_i は C_i+2（番人も含めて）である．したがって，比較と移動の総数は次の通りである．ただし，添字 min，ave，max はそれぞれ最小値，平均値，最大値を表わす．

プログラム 2.1　単純挿入法によるソーティング

```
procedure straightinsertion;
   var i,j: index;  x: item;
begin
   for i := 2 to n do
   begin x := a[i]; a[0] := x; j := i-1;
      while x .key < a[j] .key do
         begin a[j+1] := a[j]: j := j-1;
         end ;
      a[j+1] := x
   end
end
```

$$
\begin{array}{ll}
C_{\min}=n-1 & M_{\min}=2(n-1) \\
C_{\mathrm{ave}}=\frac{1}{4}(n^2+n-2) & M_{\mathrm{ave}}=\frac{1}{4}(n^2+9n-10) \\
C_{\max}=\frac{1}{2}(n^2+n)-1 & M_{\max}=\frac{1}{2}(n^2+3n-4)
\end{array}
\tag{2.4}
$$

最小回数は項目がもともと正順であった場合に得られ，最悪な場合は項目が逆順に並んでいたときに起こる．この意味で，挿入によるソーティングは，真に**自然なふるまい**を見せる．また，このアルゴリズムは**安定**なソーティングを与える．すなわち，等しいキーをもった項目の順序は不変である．

新しい項目が挿入される目的列 $a_1 \cdots a_{i-1}$ はすでに正順になっていることに着目することにより，単純挿入のアルゴリズムは容易に改良される．なぜなら，挿入点を決めるためのより高速な方法が使えるからである．2分探索は明らかにそのような方法であるが，これは，目的列を中間点でサンプルし，挿入点が見つかるまで2分割を続けるものである．この修正されたソーティング・アルゴリズムは**2分挿入法**と呼ばれ，**プログラム 2.2** で与えられる．

2分挿入法の解析　挿入位置は，$a_j.key \leq x.key < a_{j+1}.key$ のときに見つかる．したがって，探索区間長は最後には1になる．そして，これには i 個のキーからなる区間を $\lceil \log_2 i \rceil$ 回だけ2分割することが必要である．したがって，

$$C=\sum_{i=1}^{n}\lceil \log_2 i \rceil$$

この和は次の積分によって近似される．

プログラム 2.2　2分挿入法によるソーティング

```
procedure binaryinsertion;
   var i,j,l,r,m: index;   x: item;
begin
   for i := 2 to n do
   begin x := a[i]; l := 1; r := i-1;
      while l ≤ r do
      begin m := (l+r) div 2;
         if x .key < a[m] .key then r := m-1 else l := m+1
      end ;
      for j := i-1 downto l do a[j+1] := a[j];
      a[l] := x;
   end
end
```

$$\int_1^n \log x\,\mathrm{d}x = x(\log x - c)\Big|_1^n = n(\log n - c) + c \qquad (2.5)$$

ただし $c=\log e=1/\ln 2=1.44269\cdots$

比較の回数は，本質的には，項目の初めの順序には無関係である．しかしながら探索区間の分割に含まれる除算で切り捨てが行なわれるので，i 個の項目について必要となる比較回数の真の値は，期待されたものより1だけ大きいかもしれない．このかたよりの性質によって，値の小さい端での挿入位置は，高い端でのそれより平均的に少しだけ早く見つけられる．したがって，項目の初めの順序が非常に乱れているときの方が良い結果を与えることになる．事実，必要となる比較回数は項目が初めに逆順になっていたときに最小であり，正順のとき最大となる．したがって，これはソーティング・アルゴリズムの**不自然なふるまい**の例である．

$$C \doteq n(\log n - \log e \pm 0.5)$$

不幸にして，2分探索の使用によって得られる改良は，比較回数についてだけであり，必要な移動回数については改良されない．実際には，項目，すなわち，キーと関連情報の移動には2つのキーの比較より時間がかかるのが普通であり，したがってこの改良は決して十分なものではない：重要な項 M はまだ n^2 のオーダである．そして，事実すでにソートされた配列をソートしなおすには，順次的探索を用いた単純挿入法より時間がかかってしまう．この実例は，“明らかな改良”が初めに予測したほどには十分な結果をもたらさないことがしばしばあり，また場合によっては，この“改良”が改悪になりかねないことを示している．

結局，挿入によるソーティングは，ディジタル計算機にとって非常に適した方法であるようには思えない：1つの項目を挿入するたびに，項目からなる行全体を1つだけシフトするような方法は不経済である．項目の移動が1つの項目についてだけ，しかも長距離にわたって行なわれるような方法からの方がより良好な結果が期待されるであろう．このアイディアから，選択によるソーティングが導かれる．

2.2.2　単純選択によるソーティング

この方法は次の原理にもとづいている：

1. 最小のキーをもった項目を選択する．
2. これと最初の項目 a_1 とを交換する．

次に，この操作を残りの $n-1$ 個の項目について行ない，そしてその次には，残りの $n-2$ 個の項目について，というように繰り返し，最後にただ1つの項目——最大のもの——が残されるまで行なう．**表2.1**の場合と同じ8個のキーについて，この方法を示すと次のようになる．

表 2.2　単純選択によるソーティング過程の例

初期キー	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	67	94

プログラムは次のようになる．

for i := 1 **to** $n-1$ **do**
　begin “$a_i \cdots a_n$ 中の最小項目のインデックスを k に代入する．”
　　“a_i と a_k を交換する．”
　end

単純選択法と呼ばれるこの方法は，ある意味では単純挿入法の反対である：単純挿入法では，各ステップで**原列**中のただ**1つ**の次の項目および，それを挿入する場所を見つけるために，**目的列**中の**すべて**の項目について考える．一方，単純選択法では，**原列**中の**すべて**の項目を調べて最小キーをもつものをみつけ，これを**1つ**の次の項目として**目的列**に置く．単純選択法のプログラムは**プログラム2.3**で与えられる．

単純選択法の解析　明らかに，キー比較の回数 C はキー集合の初期順序に独立である．この意味で，この方法は単純挿入法に比べてより不自然なふるまいをするといえる．C は次式で与えられる．

$$C=\frac{1}{2}(n^2-n)$$

プログラム 2.3 単純選択法によるソーティング

```
procedure straightselection;
   var i,j,k: index;  x: item;
begin for i := 1 to n-1 do
      begin k := i; x := a[i];
         for j := i+1 to n do
            if a[j] .key < x .key then
            begin k := j; x := a[j]
            end ;
         a[k] := a[i]; a[i] := x;
      end
end
```

移動回数 M は，キー集合が初めに正順であるときに最小で，

$$M_{\min}=3(n-1) \tag{2.6}$$

また，それが逆順になっているとき最大である．ただし，trunc は小数点以下切捨てを意味する（訳者挿入）．

$$M_{\max}=\mathrm{trunc}\left(\frac{n^2}{4}\right)+3(n-1)$$

平均値 M_{ave} の決定は，アルゴリズムが簡単な割には困難である．その値は，数列 k_1，$\cdots$，k_n を走査したとき，先行するすべての数 k_1，$\cdots$，k_{j-1} より小さい数 k_j が見つけられる回数に依存する．この値を n 個のキーの n！個のすべての置換について平均したものは，

$$H_n-1$$

である．ここで，H_n は n 次の**調和数**である．

$$H_n=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} \tag{2.7}$$

(Knuth, Vol. I pp. 95～99 参照)．
H_n は次のように表わせる．

$$H_n=\ln n+\gamma+\frac{1}{2n}-\frac{1}{12n^2}+\cdots \tag{2.8}$$

ここで，$\gamma=0.577216\cdots$ は Euler 定数である．十分に大きな n に対しては

分数部分を省略できるので，i 番目のパスの平均代入回数は次のように近似される．

$$F_i = \ln i + \gamma + 1$$

したがって，選択ソートにおける平均移動回数 M_{ave} は F_i を $i=1$ から n まで加えたものであり，

$$M_{\mathrm{ave}} = \sum_{i=1}^{n} F_i = n(\gamma+1) + \sum_{i=1}^{n} \ln i$$

離散項の和をさらに積分で近似すると，

$$\int_1^n \ln x \, \mathrm{d}x = x(\ln x - 1) \Big|_1^n = n \ln n - n + 1$$

したがって，次の近似値を得る．

$$M_{\mathrm{ave}} \doteqdot n(\ln n + \gamma) \qquad (2.9)$$

一般には，単純選択のアルゴリズムは単純挿入より好ましいと結論することができるが，キー集合が初めから正順であったり，あるいはほぼ正順であるときには単純挿入法の方がまだいくらか速い．

2.2.3　単純交換によるソーティング

ソーティング方法は，完全に明確に類別されるということはほとんどない．先に述べた2つの方法は，いずれも交換ソートと見ることもできる．しかしながら，この節では，2つの項目の交換がそのプロセスの主たる特徴であるような方法を示す．これから述べる単純交換のアルゴリズムは，すべての項目がソートされるまで，隣り合う2つの項目を比較し**交換する**という原理に基づくものである．

先の単純選択法のように，現在残っている集合中の最小の項目を配列の左端に向かってシフトさせるというパスを配列に対して繰り返す．配列が水平方向ではなく垂直方向に延びているように見方を変え，多少の想像力を働かせて，項目をそのキーに応じた重さをもった水槽中の泡 (bubble) と考える．そうすると，配列に対する各パスは，泡をその重さに応じた適当なレベルまで上昇させる結果となる (**表 2.3** 参照).
この方法は，**バブル・ソート** (Bubblesort) として広く知られている．その最も簡単な形が**プログラム 2.4** に示されている．

表 2.3 バブル・ソートの例

初期配列	i=2	i=3	i=4	i=5	i=6	i=7	i=8
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

プログラム 2.4 バブル・ソート

```
procedure bubblesort;
   var i,j: index;   x: item;
begin for i := 2 to n do
      begin for j := n downto i do
            if a[j−1] .key > a[j] .key then
            begin x := a[j−1]; a[j−1] := a[j]; a[j] := x
            end
      end
end {bubblesort}
```

このアルゴリズムは容易に改良できる．表 2.3 の例題では，最後の 3 回のパスは項目の順序に影響がないことを示している．なぜならば，項目は既にソートされているからである．このアルゴリズムを改良するための明らかな方法は，あるパス中で交換が起こったか否かを記憶し，もし交換が起こらなかったときにはアルゴリズムを停止させることである．しかしながら，この改良されたアルゴリズムもまた交換が起こったという事実を記憶するだけでなく，最後の交換の位置（インデックス）を記憶することにより，さらに改良することができる．たとえば，このインデックス k よりも下にあるすべての隣り合う対が望ましい順序になっていることは明白である．したがって，次の走査はこのインデックスで停止してよく，前もって決めた下限 i まで行なう必要はない．しかしながら，注意深いプログラマなら奇妙な非対称性に気づくであろう：ソートされた配列の "heavy" end（水槽の底）に間違って置かれた泡は，1 回のパスで正しい位置に移動されるが，"light" end（水面）に間違って置かれた項目は，正しい位置に向かって，各パスで 1 ステップずつしか沈まない．たとえば，配

列

12　18　42　44　55　67　94　06

は，改良されたバブル・ソートによると1回のパスでソートされるが，配列

プログラム2.5　シェーカ・ソート

```
procedure shakersort;
   var j,k,l,r: index;   x: item;
begin l := 2; r := n; k := n;
   repeat
      for j := r downto l do
         if a[j-1] .key > a[j] .key then
         begin x := a[j-1]; a[j-1] := a[j]; a[j] := x;
            k := j
         end ;
      l := k+1;
      for j := l to r do
         if a[j-1] .key > a[j] .key then
         begin x := a[j-1]; a[j-1] := a[j]; a[j] := x;
            k := j
         end ;
      r := k-1;
   until l > r
end {shakersort}
```

表2.4　シェーカ・ソートの例

l = 2	3	3	4	4
r = 8	8	7	7	4
↑	↓	↑	↓	↑
44	06	06	06	06
55	44	44	12	12
12	55	12	44	18
42	12	42	18	42
94	42	55	42	44
18	94	18	55	55
06	18	67	67	67
67	67	94	94	94

94 06 12 18 42 44 55 67

のソートには7回のパスを必要とする．この不自然な非対称性は，第3の改良，すなわち連続するパスの方向の逆転を示唆する．この改良アルゴリズムは，**シェーカ・ソート** (shakersort) と呼ぶのが適当であろう．表2.3で使われたのと同じ8個のキーに対して，この方法を適用したときの様子が表2.4に示されている．

バブル・ソートとシェーカ・ソートの解析　単純交換アルゴリズムの比較回数は，

$$C=\frac{1}{2}(n^2-n) \tag{2.10}$$

であり，移動（項目の代入）回数の最小，平均および最大値は次の通りである．

$$M_{\min}=0,\quad M_{\mathrm{ave}}=\frac{3}{4}(n^2-n),\quad M_{\max}=\frac{3}{2}(n^2-n) \tag{2.11}$$

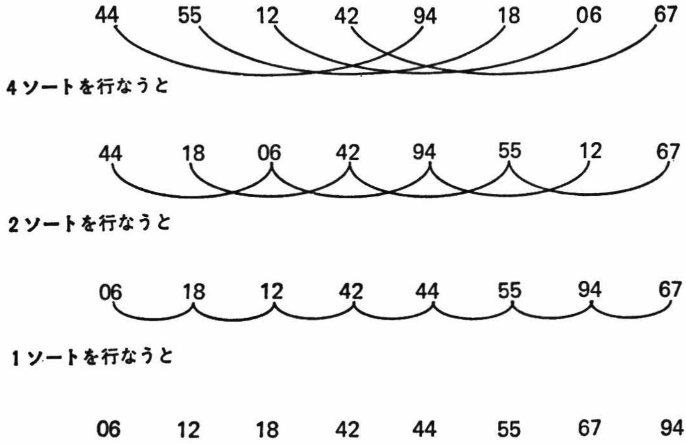
改良された方法，特にシェーカ・ソートの解析は複雑である．比較回数の最小値は，$C_{\min}=n-1$ である．Knuth は，改良されたバブル・ソートに対して，パスの平均回数が $n-k_1\sqrt{n}$ に比例し，平均比較回数が $\frac{1}{2}[n^2-n(k_2+\ln n)]$ に比例するという結果を得ている．しかしながら，上で述べたすべての改良は，交換回数には全く影響がないことに注目しよう．それらは，ただ冗長な2重チェックの数を減らしたに過ぎない．不幸なことに，2つの項目の交換は，キーの比較よりもコストのかかる操作であるのが普通である．したがって，われわれの巧妙な改良も直感的に期待していた程には大きな効果をもたらさない．

この解析は，交換ソートやそのちょっとした改良が，挿入および選択ソートより劣っていることを示している．実際バブル・ソートには，その面白くて覚えやすい名前を除いては，ほとんど長所がない．シェーカ・ソートは，項目が既にほとんど正順なっている——実際には稀なケースだが——ことがわかっている場合に使うのなら利点がある．

ソートの過程で，n 個の項目の各々が移動しなければならない平均距離は $n/3$ であることを示すことができる．この数字は改良された，すなわちもっと効果的なソーティング法を探す場合の手掛りとなる．すべての単純ソーティング法は，各基本ステップで本質的に各項目の位置を1だけしか動かしていない．

したがって，これらの方法では n^2 のオーダのステップ数がどうしても必要である．どのような改良も，1回の操作で項目をより長距離動かすという原理

表 2.5　減少する増分による挿入ソート



に基づくものでなければならない．

以下では，3つの改良された方法について議論するが，それらは各々基本的ソーティング法，すなわち挿入，選択および交換法に対応するものである．

2.2.4　減少する増分による挿入ソート

単純挿入ソートの改良が D. L. Shell によって 1959 年に提案された．8個の項目からなるわれわれの標準的例題を使ってこの方法を説明する（**表 2.5**参照）．まず初めに，4つだけ離れたすべての項目をグループにまとめて別々にソートする．この過程を4ソートという．8個の項目から成るこの例では，各グループはちょうど2個の項目を含んでいる．この最初のパスの次には，2だけ離れた項目を再びグループにまとめて新たにソートする．このプロセスは2ソートと呼ばれる．最後に，第3番目のパスですべての項目は，通常のソートまたは1ソートされる．

すべての項目が関係するソートを何回も行なうことが必要なので，節約される以上の作業量が余分にいるのではないかと最初は考えるかもしれない．しかしある鎖についてのソートでは，比較的少数の項目しか関係しないか，あるいは項目は既にかなり良く正順になっているので，比較的少数の並び換えしか必要としない．

この方法によって，正順の配列が得られることは明白であり，また，各パスはそれ以前のパスから利益を受けていることも全く明らかである（なぜなら各

i ソートでは，その前の $2i$ ソートでソートされた2つのグループを結合しているからである)．また，その最後の増分が1であるならば，どのような増分の系列も受け入れられるものであることも明白である．なぜなら最悪の場合でも，最後のパスがすべての仕事をしてくれるからである．しかし，2のべき乗以外の減少する増分を用いた方が良い結果を生むということは余り明白ではない．

したがって，プログラムは特定な増分列に依存しない形に作られている．t 個の増分を次のように表わす．

$$h_1,\ h_2, \cdots,\ h_t$$

ただし，

$$h_t=1, \quad h_{i+1}<h_i \tag{2.12}$$

各 h ソートは，単純挿入ソートによってプログラムが作られているが，番人(sentinel) 法を使うことにより，挿入場所の探索の停止条件を簡単にしている．

各ソートは自分自身の番人を置く必要があり，またその位置を決定するためのプログラムはできるだけ簡単でなければならないことは明らかである．したがって，配列 a は1つの成分 $a[0]$ のみでなく，h_1 個の成分だけ拡張されなければならず，したがって次のように宣言される．

$$a\text{: }\mathbf{array}\ [-h_1\,..\,n]\ \mathbf{of}\ \mathit{item}$$

$t=4$ に対する**プログラム 2.6** では，このアルゴリズムは *shellsort* [2-11] と呼ばれる手続きによって記述されている．

Shell ソートの解析　このアルゴリズムの解析には，非常に困難な数学的問題がいくつかあり，それらのうちの多くはまだ解かれていない．特に，どのような増分の選択が最良の結果を与えるかは知られていない．しかしながら，ひとつの驚くべき事実は，それらは互いに倍数であってはいけないということである．これは，各ソーティング・パスで，それまで全く干渉のなかった2つの鎖を結合するという，上に挙げた例からも明らかな現象を避けようとするものである．種々の鎖の間の干渉ができるだけ頻繁に起こることが非常に望ましい．また，次の定理が成立する：

k ソートされた列を i ソートしても，それは k ソートされている．

プログラム 2.6　Shell ソート

```
procedure shellsort;
    const t = 4;
    var i,j,k,s: index;  x: item; m: 1 .. t;
        h: array [1 .. t] of integer;
begin h[1] := 9; h[2] := 5; h[3] := 3; h[4] := 1;
    for m := 1 to t do
    begin k := h[m]; s := -k; {sentinel position}
        for i := k+1 to n do
        begin x := a[i]; j := i-k;
            if s=0 then s := -k; s := s+1; a[s] := x;
            while x .key < a[j] .key do
            begin a[j+k] := a[j]; j := j-k
            end ;
            a[j+k] := x
        end
    end
end
```

Knuth [2-8] は，増分は次のように選ぶのが理にかなっているという証拠を示している．

$$1, 4, 13, 40, 121, \cdots \quad \text{(逆順で)}$$

ここで，$h_{k-1}=3h_k+1, h_t=1$，かつ $t=\lfloor \log_3 n \rfloor -1$ である．彼はまた次の数列

$$1, 3, 7, 15, 31, \ldots$$

を推奨している．ここで，$h_{k-1}=2h_k+1, h_t=1$，かつ $t=\lfloor \log_2 n \rfloor -1$ である．後者の選択については，数学的解析により，Shell ソート・アルゴリズムを用いて n 個の項目をソートするのに $n^{1.2}$ に比例する作業量が必要であることが求められている．これは n^2 に対する重要な改良ではあるが，より良いアルゴリズムが知られているので，これ以上この方法について説明するつもりはない．

2.2.5　木ソート

単純選択によるソーティング法は，n 個の項目から最小キーを選び，次に残りの $n-1$ 個の項目から最小キーを，…というように選択を繰り返すことに基づいている．明らかに，n 個の項目中から最小キーを見つけるには，$n-1$ 回の

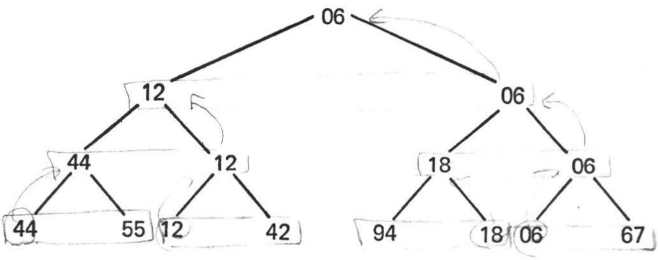


図 2.3 2つのキー間の選択の繰返し

比較が必要であり，$n-1$ 個の項目中からそれを見つけるには，$n-2$ 回の比較が必要である．そこで，どのようにすればこの選択ソートを改良できるであろうか．それには，1回の走査で最小項目ひとつを見つけるだけではなく，より多くの情報を求めておくことが必要である．たとえば，$n/2$ 回の比較によって，各項目対† のうちの小さいキーを決めることができるし，さらに $n/4$ 回の比較によって，それらの小さいキーからなる各対のうちのより小さいキーを選択することができる．そして，この操作をさらに続けると，結局 $n-1$ 回の比較によって，**図 2.3** で示されるような選択木を構成することができ，最小キーはその根として求められる [2-2].

次のステップでは，最小キーによってマークされた道を下り最下点（端点）まで行き，これを穴（またはキー ∞）によって置き換える．そして，再びこの道を根に向かって戻りながら，各節点の値を順次その子供の節点の値の最小値で置き換える．以上のようにして最小キーを消去する．

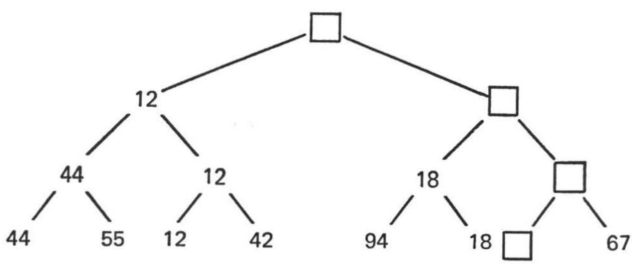


図 2.4 最小キーの選択

† 訳注　n 個の項目から重複を許さずに作った $n/2$ 個の項目の対

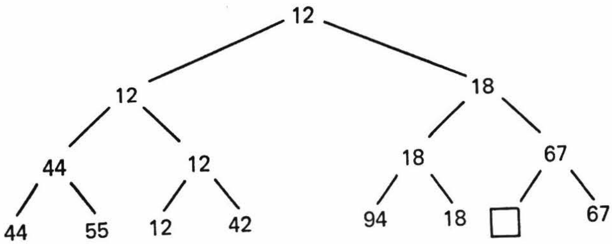


図 2.5　穴の詰め直し

この木の根に現われた項目は（今では2番目に）最小のキーであり，再び消去することができる．このような選択ステップをn回繰り返すことにより，木は空になり（すなわち穴だけ）ソーティング過程は終了する．n 個の各選択ステップは $\log_2 n$ の比較しか必要としないことに注意しよう．したがって，全選択過程では $n \log_2 n$ のオーダの基本操作と木の構成に必要な n ステップしか必要としない．これは n^2 ステップを必要とする単純法や，さらには $n^{1.2}$ ステップを必要とする Shell ソートに比べても非常に重要な改良である．

当然，管理作業はより精巧になっており，したがって木ソートでは，個々のステップはより複雑である：最初のパスから得られた大量のデータを保持しておくためには，ある種の木構造が作られなければならない．われわれの次の仕事は，この情報を効率的に編成する方法を見つけることである．

遂には木全体をおおってしまい，そして多くの無駄な比較の原因である穴の必要性をなくすことが特に望ましいことはもちろんである．さらに，n個の項目からなる木を表現するのに，上のように $2n-1$ 個の記憶セルを用いるのではなく，n 個で表現する方法を見つけるべきである．これらの目標は，その発明者である J. Williams [2-14] によって**ヒープ・ソート** (heapsort) と呼ばれた方法により実際に達成された．この方法がより伝統的な木ソート法の著しい改良であることは明らかである．

ヒープ (heap) は，すべての $i=l, \cdots, r/2$ に対して，

$$h_i \leq h_{2i}$$
$$h_i \leq h_{2i+1} \qquad (2.13)$$

を満たすキーの列

$$h_l, h_{l+1}, \cdots, h_r$$

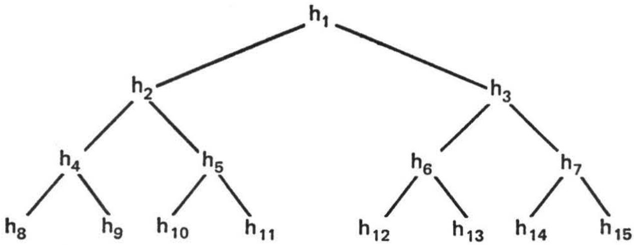


図2.6　2分木と考えた配列

として定義される．2分木が図2.6で示されるような配列として表現されるならば，図2.7と2.8のソート木はヒープであるということになり，特にヒープの要素 h_1 はその最小要素である．

$$h_1=\min(h_1 \dots h_n)$$

ある値 l と r に対して要素 $h_{l+1} \cdots h_r$ をもつヒープが与えられたとき，新しい要素 x を追加して延張したヒープ $h_l \dots h_r$ を作ると仮定しよう．たとえば，図2.7で示される初期ヒープ $h_1 \cdots h_7$ を考え，要素 $h_1=44$ によってこのヒープを左に延長する．新しいヒープは次のようにして得られる．まず x を木の根に置き，x から出る2つの子供の節点のうちその値が小さい方のものと x を比較する．もし x がその値より大きいなら，x とその値と交換することにより x をシフト・ダウンさせる．この操作を可能な限り繰り返すことにより，新しいヒープが作られる．

与えられた例では，値44は最初に06と，次に12と交換され，そして図2.8で示されるような木になる．

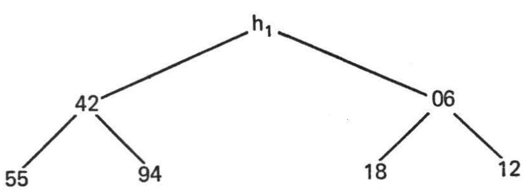


図2.7　7個の要素をもつヒープ

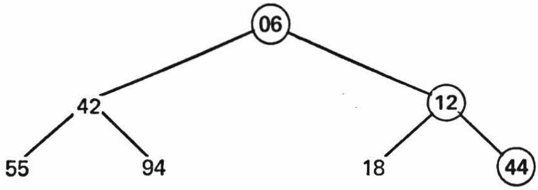


図2.8　キー44のヒープ中のシフト

このシフト・アリゴリズムを次のように定式化する．ここで，i, j は各シフト・ステップ中で交換される項目を示すインデックス対である．読者は提案されたこのシフト法がヒープを定義している条件（2.13）を保存することをぜひ確認して欲しい．

“その場所で”ヒープを構成する簡単な方法が R. W. Floyd によって提案された．これは，**プログラム2.7** で示されるシフト手続きを用いる．配列 $h_1 \cdots h_n$ が与えられたとき，明らかに要素 $h_{n/2+1} \cdots h_n$ は既にヒープを成している．なぜなら，$j = 2i$（または $j = 2i+1$）であるような2つのインデックス i, j が存在しないからである．これらの要素は，対応する2分木（図2.6参照）の最下行であると考えられるが，それらの間にはいかなる順序関係も要求されない．次に各ステップで新しい要素を1つずつ追加し，それをシフトによって正しく位置づけることによりこのヒープを左に延長する．このプロセスは**表2.6** で示され，表2.6のヒープが得られる．

プログラム2.7　シ　フ　ト

```
procedure sift(l,r: index);
    label 13;
    var i,j: index;  x: item;
begin i := l; j := 2*i; x := a[i];
    while j ≤ r do
    begin if j < r then
            if a[j] .key > a[j+1] .key then j := j+1;
        if x .key ≤ a[j] .key then goto 13;
        a[i] := a[j]; i := j; j := 2*i {sift}
    end;
13: a[i] := x
end
```

表 2.6 ヒープの構成

44	55	12	42	94	18	06	67
44	55	12	42	94	18	06	67
44	55	06	42	94	18	12	67
44	42	06	55	94	18	12	67
06	42	12	55	94	18	44	67

したがって，n 個の要素 $h_1 \cdots h_n$ をもつヒープをその場所で生成するプロセスは次のように書かれる．

```
l := (n div 2) + 1;
while l > 1 do
    begin l := l-1; sift(l,n)
    end
```

ソートされた要素を得るためには，n 回のシフト・ステップを実行し，各ステップの後にヒープの先頭要素を取り出さなければならない．ここで再び，先頭に現われた要素をどこに格納するか，そして“その場所で”のソートが可能かどうかという問題が生じてくる．もちろんその解はある！ 各ステップで最後の要素 x をヒープから取り出し，ヒープの先頭の要素をいま空いた x の場所に格納する．そして，x を適当な場所にシフト・ダウンする．**表 2.7** に必要となる $n-1$ 回のこのステップが示されている．

表 2.7 ヒープ・ソート過程の例

06	42	12	55	94	18	44	67
12	42	18	55	94	67	44	06
18	42	44	55	94	67	12	06
42	55	44	67	94	18	12	06
44	55	94	67	42	18	12	06
55	67	94	44	42	18	12	06
67	94	55	44	42	18	12	06
94	67	55	44	42	18	12	06

このプロセスは，手続き *shift* (プログラム 2.7) を用いて次のように書かれる．

```
r := n;
while r > 1 do
   begin x := a[1]; a[1] := a[r]; a[r] := x;
      r := r-1; sift(1,r)
   end
```

表 2.7 は結果の順序が実際には逆転していることを示している．しかし，これはシフト手続き中の順序関係の向きを変えることにより容易に直すことができる．以上より**プログラム 2.8** で与えられる手続き *heapsort* が得られる．

プログラム 2.8　ヒープ・ソート

```
procedure heapsort;
   var l,r: index;  x: item;
   procedure sift;
      label 13;
      var i,j: index;
   begin i := l; j := 2*i; x := a[i];
      while j ≤ r do
      begin if j < r then
                 if a[j] .key < a[j+1] .key then j := j+1;
         if x .key ≥ a[j] .key then goto 13;
         a[i] := a[j]; i := j; j := 2*i
      end ;
   13: a[i] := x
   end ;
begin l := (n div 2) + 1; r := n;
   while l > 1 do
      begin l := l-1; sift
      end ;
   while r > 1 do
      begin x := a[1]; a[1] := a[r]; a[r] := x;
         r := r-1; sift
      end
end {heapsort}
```

ヒープソートの解析　一見したところ，このソーティング法がよい結果を与えるということは明らかではない．なぜなら，多数の項目をまず左側にシフトし，次にそれらを遠く右側にたい積させるからである．実際，例で示されたような少数の項目に対しては，この方法は推奨できない．しかし，大きな n に対してはヒープ・ソートは大変効率的であり，さらに，Shell ソートと比較した場合，n が大きければ大きい程有利となる．

最悪の場合には，それぞれ $\log(n/2)$，$\log(n/2+1)$，…，$\log(n-1)$ 回だけ項目をシフトする $n/2$ 回のシフト・ステップが必要である．ここで，対数の底は 2 であり小数部は切捨てられる．次に，ソーティング段階で $n-1$ 回のシフトが行なわれるが，それには，高々 $\log(n-1)$，$\log(n-2)$，…，1 回の移動が必要である．この議論から，ヒープ・ソートは，最悪の場合でも $n \log n$ のオーダのステップ数しか必要としないことがわかる．最悪の場合のこの優れた性能は，ヒープ・ソートの最も強力な性質である．

最悪（または，最良）な結果がどのような場合に起こるかは全く明らかではない．しかし，一般にはヒープ・ソートは項目が多少とも逆順になっている初期列を好む傾向があり，したがって，不自然なふるまいを見せる．明らかに，もし逆順になっていればヒープ生成段階では移動を必要としない．われわれの 8 個の項目からなる例については，次の初期列が移動の最小，最大値を与える．

94　67　44　55　12　42　18　6　に対して $M_{\min}=13$

18　42　12　44　6　55　67　94　に対して $M_{\max}=24$

移動の平均値は，近似的には $\frac{1}{2}n\cdot\log n$ であり，この値からのずれは比較的小さい．

2.2.6 分割ソート

挿入と選択の原理に基づく高度なソーティング法を 2 つ述べたので，次に交換の原理に基づいた第 3 番目の改良された方法を導入する．バブル・ソートは，3 つの単純ソーティング・アルゴリズムの中では平均的に最も性能が悪いことを考えると，比較的大きな改良度が期待される．それでも，ここで述べる交換法の改良が，配列のソーティング法としてこれまで知られているもののうちで最も良い方法を生み出したということは驚きである．その性能が余りに劇的であったので，その発明者である C. A. R. Hoare はそれをクィック・ソート (Quicksort) と名づけた [2-5 および 2-6].

クィック・ソートは，交換は長距離にわたって行なうのが最も効果的であるという事実に基づいている．キーが逆順になっている n 個の項目があるとしよう．このとき，まず左端と右端を交換し，次に両端から徐々に内側に進むことにより，それらは $n/2$ 回だけの交換を行なうことによってソートすることができる．もちろんこれが可能なのは，その順序が正確に逆順であることがわかっているときだけであるが，それでもこの例題から何かを学びとることができる．次のアルゴリズムを考えてみよう：ランダムに任意の項目を取り出し，これを x とする．項目 $a_i>x$ が見つかるまで配列を左から走査する．次に，項目 $a_j<x$ が見つかるまで右から走査する．そしてこの2つの項目を交換し，この“走査と交換”プロセスを2つの走査が配列中のどこかで出会うまで実行する．この結果，配列は x より小さいキーをもった左部分と，x より大きいキーをもった右部分に分割される．この分割プロセスは，**プログラム 2.9** の手続きのように定式化される．ただし，関係 $>$ と $<$ は $\geq$ と $\leq$ によって置き換えられ，その否定 $<$ と $>$ が **while** 節で使われている．この変更により，x は2つの走査に対する番人として働く．

たとえば，比較相手 x として中央値42を選んだとすると，キーの配列

44　55　12　*42*　94　06　18　67

を分割された配列

18　06　12　|42|　94　55　44　67

プログラム 2.9　分　割

```
procedure partition;
var w,x: item;
begin i := 1; j := n;
    select at random an item x;
    repeat
        while a[i].key < x .key do i := i+1;
        while x.key < a[j] .key do j := j-1;
        if i ≤ j then
            begin w := a[i]; a[i] := a[j]; a[j] := w;
                i := i+1; j := j-1
            end
    until i > j
end
```

にするには2回の交換が必要であり，最終のインデックス値は $i=5$ と $j=3$ である．キー $a_1, \ldots, a_{i-1}$ は $x=42$ より小さいかまたは等しく，キー $a_{j+1}, \cdots, a_n$ は x より大きいかまたは等しい．したがって，2つの分割部分，すなわち

$$\begin{aligned} &a_k.key \leq x.key \quad ただし\ k=1, \cdots, i-1 \\ &a_k.key \geq x.key \quad ただし\ k=j+1, \cdots, n \end{aligned} \tag{2.14}$$

があり，また，

$$a_k.key = x.key \quad ただし\ k=j+1, \cdots, i-1$$

である．このアルゴリズムは単純であり，また，本質的な比較対象 i, j および x を高速レジスタに置いておけるので能率がよい．しかし，n 個のキーが同一の場合には $n/2$ 回の交換が必要であり厄介である．この不必要な交換は，走査ステートメントを次のように変えれば除くことができる．

while $a[i].key \leq x.key$ **do** $i := i+1$;
while $x.key \leq a[j].key$ **do** $j := j-1$;

しかしこの場合には，配列のメンバーである選択要素 x は2つの走査のための番人にはもはやなりえない．そして，より複雑な停止条件を用いなければ，同一のキーから成る配列では走査は配列の限界をはみ出してしまう．したがって，プログラム2.9で使われている簡潔な条件は，平均的に“ランダム”な場合には比較的まれにしか起こらない余分の交換を補って余りある．しかし，交換ステップを制御している条件節を $i \leq j$ から

$$i < j$$

に変更するこしにより，わずかながら節約が可能である．しかし，この変更は2つの文

$$i := i+1;\ j := j-1$$

の上にまで及んではならず，したがって別々の条件節が必要になる．

この節 $i \leq j$ の必要性は $x=2$ とした次の例によって示すことができる．

1 1 1 *2* 1 1 *1*

初めの走査により

1 1 1 1 1 1 2

および，$i=5$, $j=6$ となる．次の走査では配列は不変であり $i=7$, $j=6$ に

なる．もし，交換が条件$i \leq j$のもとで行なわれないとすると，a_6とa_7が間違って交換されてしまうであろう．

分割アルゴリズムの正しさは，2つの言明 (2.14) が repeat 文の不変式であることを証明することにより確認できる．初めに$i=1$，$j=n$についてはそれらは明らかに正しい．そして，$i>j$が成立して脱け出るときには，それらは望ましい結果を意味する．

さて，われわれの目標は，単に項目からなる原配列の分割を見つけることだけではなく，それをソートすることであることを思い出そう．しかし，分割からソートへはほんの少しのステップである．すなわち配列を分割した後で，同じ手続きを2つの部分に適用して再分割を行ない，ただ1つの項目だけからなる分割が得られるまでこの操作を繰り返せばよい．このアルゴリズムは**プログラム2.10**に与えられている．

手続き *sort* は自分自身を再帰的に起動する．アルゴリズムにおけるこのような再帰の使用は非常に強力な道具であり，3章でさらに論じられるであろう．昔設計されたプログラム言語のなかには，ある技術的な理由によって再帰が許されないものがある．したがって，次に同じアルゴリズムが非再帰的手続きとし

プログラム 2.10　クィック・ソート

```
procedure quicksort;
    procedure sort (l,r: index);
        var i,j: index;  x,w: item;
    begin i := l; j := r;
        x := a[(l+r) div 2];
        repeat
            while a[i] .key < x .key do i := i+1;
            while x .key < a[j] .key do j := j-1;
            if i <= j then
            begin w := a[i]; a[i] := a[j]; a[j] := w;
                i := i+1; j := j-1
            end
        until i > j;
        if l < j then sort(l,j);
        if i < r then sort(i,r)
    end ;
begin sort(1,n)
end {quicksort}
```

てどのように表わされるかを示そう．明らかに，このためには再帰を繰返しによって表現しなければならないので，ある種の付加的な管理作業が必要になる．

繰り返しを用いた解の要点は，まだ実行してない分割要求をリストとして管理することにある．各ステップの実行後には分割作業が2つできるが，次の繰返しで実行できるのは，そのうちの1つだけである．もう1つのものはそのリストにスタックしておく．したがって，要求のリストが特定の順番，すなわち逆の順番に満足させられることはもちろん本質的である．これは，一番初めにリストされた要求は一番最後に満たされ，そしてその逆も成立することを意味する．次に示すクィック・ソートの非再帰版では，各要求は，さらに分割されるべき部分の範囲を示す左および右インデックスによって表現されている．このために，*stack* という名前の配列と，その最も新しいエントリを示すインデックス s を導入する（**プログラム 2.11** 参照）．スタックの長さ m をどのように選ぶべきかはクィック・ソートの解析のところで議論されるであろう．

クイック・ソートの解析　　クィック・ソートの性能を解析するためには，まず，分割プロセスのふるまいを調べる必要がある．境界値 x を選択した後で全配列が走査されるので，ちょうど n 回の比較が行われる．一方，交換の回数は次のような確率論的議論によって決定できる．

分割されるべきデータ集合が，n 個のキー $1, \cdots, n$ から構成され，境界値として x を選択したと仮定しよう．分割操作の後では，x は配列中の x 番目の位置を占める．実行される交換の回数は，左部分の長さ $(x-1)$ に交換されたキーの生起確率を掛けたものである．キーは，境界値 x より小さくないとき交換され，したがって，この確率は $(n-x+1)/n$ である．ゆえに，交換回数の期待値は，この交換回数をすべての可能な境界値について加え n で割ることにより次のように得られる．

$$M=\frac{1}{n}\sum_{x=1}^{n}\frac{x-1}{n}\cdot(n-x+1)=\frac{n}{6}-\frac{1}{6n} \qquad (2.15)$$

したがって，交換回数の期待値は，近似的に $n/6$ である．

もし，非常に幸運にも，境界値としていつも中央値を選択できたとしたら，各分割プロセスは配列を半分に分けるので，ソートするのに必要なパスの回数は $\log n$ である．したがって，比較の全回数は $n \log n$ であり，交換の全回数は $n/6 \cdot \log n$ である．もちろんいつでも中央値を選べるとは限らない．事実，そのようなチャンスは，たったの $1/n$ である．しかし，驚いたことには，境

プログラム2.11　クィック・ソートの非再帰版

```
procedure quicksort 1;
   const m = 12;
   var i,j,l,r: index;
      x,w: item;
      s: 0 .. m;
      stack: array [1 .. m] of
               record l,r: index end;
begin s := 1; stack[1] .l := 1; stack[1] .r := n;
   repeat {take top request from stack}
      l := stack[s] .l; r := stack[s] .r; s := s-1;
      repeat {split a[l] ... a[r]}
         i := l; j := r; x := a[(l+r) div 2];
         repeat
            while a[i] .key < x .key do i := i+1;
            while x .key < a[j] .key do j := j-1;
            if i ≤ j then
            begin w := a[i]; a[i] := a[j]; a[j] := w;
               i := i+1; j := j-1
            end
         until i > j;
         if i < r then
         begin {stack request to sort right partition}
            s := s+1; stack[s] .l := i; stack[s] .r := r
         end ;
         r := j
      until l ≥ r
   until s = 0
end {quicksort 1}
```

界値がランダムに選ばれたとしても，クィック・ソートの平均的性能は，最適な場合に比べて 2・ln 2 分の 1 に低下するだけである.†

しかし，クィック・ソートにも落し穴がある. まず第1に，すべての高度な方法と同様に n の小さい値については，中位の性能しかないことである. 他の高度な方法より良い点は，小さい分割を扱うのに単純なソーティング法を組み込み易いことである. これは，その再帰的プログラムを考えるときには特に有

† 訳注　2 ln 2=1.3629

利である.

しかし，最悪な場合の問題はまだ残っている．この場合クィック・ソートの性能はどうであろうか．残念ながら，その答は期待にそむくものであり，クィック・ソートのひとつの弱点を明らかにするものである（このような場合にはslowsort になる)．たとえば，比較相手 x として分割部分の最大値がいつも選ばれるという不運な場合を考えてみよう．このときには，n 個の項目からなる配列は，$n-1$ 個の項目からなる左部分と，1個の項目からなる右部分に分けられる．この結果，$\log n$ 回ではなく n 回の分割が必要になり，ゆえに最悪な場合の性能は n^2 のオーダになる.

明らかに，決定的なステップは比較相手 x の選択である．われわれの例題プログラムでは，配列の中間要素が選ばれていた．最初または最後の要素 $a[l]$, $a[r]$ を選んでもほぼ同様に良いことに注意しよう．このときには，初めに配列がソートされているときが最悪の場合であり，クィック・ソートは自明なケースを嫌い，順序の乱れた配列を好む傾向を見せる．中間要素を選ぶときには，クィック・ソートのこの奇妙な特徴は余り明らかではない．なぜなら，この場合には初めに配列がソートされているときが最適となるからである！　事実，中間位置が選ばれた場合の方が平均的性能はわずかではあるが良くなる．Hoare は，x をランダムに選択する，あるいは，たとえば3個程度のキーからなる小数のサンプルの中央値を x として選ぶことを提案している [2-12 および 2-13]．そのような賢明な方法によっても，クィック・ソートの平均的性能はほとんど影響されないが，最悪な場合の性能は非常に改善される．明らかに，クィック・ソートによるソーティングは，運が悪いときにはどれ位の損失が許されるかを知っていなければならないような賭けに似ているところがある.

以上の経験から学ばねばならないことが1つあるが，それはプログラマに直接関係することである．上で述べたプログラム 2.11 の性能が最悪になる場合のふるまいから，何が導かれるであろうか．われわれは，各分割からただ1個の要素のみからなる右部分が生じるこしを知った．そして，この部分をソートする要求はスタックされ，後で実行される．したがって，要求の最大数，すなわち必要となるスタックの長さは n である．もちろん，これは全く受け入れられないものである．(再帰的プログラムの場合についても決して良いとはいえない，というより，実際に悪い．なぜなら，手続きの再帰的呼出しを許すようなシステムでは，手続き呼出しに際して局所変数やパラメータが自動的に格納され，そのためにスタックが陰で使われるからである)．これを避けるためには，大きい方の分割部分のソート要求をスタックし，小さい部分の再分割を続けて

実行すればよい．この場合には，スタックの長さmは，$m = \log_2 n$に制限することができる．

プログラム2.11に必要な変更は，新しい要求を準備する部分だけに限られ，次のようになる．

```
if j−l < r−i then
begin if i < r then
    begin {stack request for sorting right partition}
        s := s+1; stack[s] .l := i; stack[s] .r := r
    end;
    r := j  {continue sorting left partition}                (2.16)
end else
begin if l < j then
    begin {stack request for sorting left partition}
        s := s+1; stack[s] .l := l; stack[s] .r := j
    end;
    l := i  {continue sorting right partition}
end
```

2.2.7　中央値を見つける方法

n個の項目の中央値は，それらのうちの半分より小さく（または等しく）かつ，他の半分より大きい（または等しい）ような項目として定義される．たとえば，次の数列の中央値は18である．

16　12　99　95　18　87　10

中央値を決定する1つの確実な方法は，そのn個の項目をソートし中間の項目を取り出すことであるので，中央値を見つける問題は，ソーティングの問題と関連づけられることが多い．しかし，プログラム2.9による分割を使えば，もっと速く中央値を見つけられる可能性がある．この方法をn個の項目中のk番目に小さいものを見つける問題に一般化することは容易であり，このときには，中央値を見つけることは，$k = n/2$という特殊な場合に当る．

C. A. R. Hoare [2-4]によって考察されたこのアルゴリズムの動作は次のようである．まず，$l = 1$，$r = n$，および，分割（境界）値xとして$a[k]$を選んでクイック・ソートの分割操作を行なう．その結果，インデックス値i, jは次のようになる．

1. $a[h] \leq x$　ただし　$h < i$

2. $a[h] \geq x$ ただし $h > j$ (2.17)
3. $i > j$

ここで，3つの可能な場合がある．

1. 境界値 x が小さ過ぎる：その結果，2つの分割部分の境界は望ましい値 k より下にある．したがって，要素 $a[i] \cdots a[r]$ つにいて分割プロセスを繰り返さなければならない（図2.9参照）．

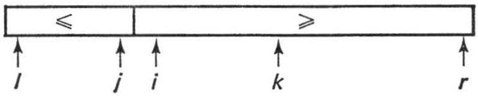


図2.9　境界値が小さ過ぎる場合

2. 境界値 x が大き過ぎる：分割操作を部分 $a[l] \ldots a[j]$ に対して繰り返さなければならない（図2.10参照）．

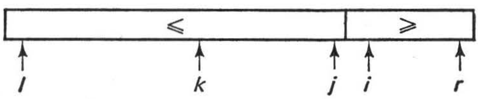


図2.10　境界値が大き過ぎる場合

3. $j < k < i$：要素 $a[k]$ は定められた割合に配列を2つの部分に分割している．したがって，これは求める分位数 (quantile) である（図2.11参照）．

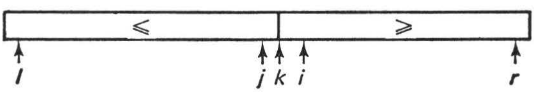


図2.11　境界値が正しい場合

場合3. になるまで分割プロセスを繰り返さなければならないが，この繰返しは次のプログラムによって表現される．

```
l := 1; r := n;                                    (2.18)
while l < r do
    begin x := a[k];
        partition(a[l] . . . a[r]);
        if j < k then l := i;
        if k < i then r := j
    end
```

このアルゴリズムの正当性の形式的証明については，Hoareによる原論文を参照して欲しい．プログラム *find* 全体は，これから直ちに導かれる．

プログラム 2.12　*k* 番目の要素の発見

```
procedure find (k; integer);
    var l,r,i,j,w,x: integer;
begin l := 1; r := n;
    while l < r do
    begin x := a[k]; i := l; j := r;
        repeat {split}
            while a[i] < x do i := i+1;
            while x < a[j] do j := j-1;
            if i ≤ j then
                begin w := a[i]; a[i] := a[j]; a[j] := w;
                    i := i+1; j := j-1
                end
        until i > j;
        if j < k then l := i;
        if k < i then r := j
    end
end {find}
```

もし，各分割によって，求める分位数の存在する部分の長さが平均的に2等分されると仮定すれば，必要な比較回数は，

$$n+\frac{n}{2}+\frac{n}{4}+\ldots+1 \doteq 2n \tag{2.19}$$

すなわち，オーダ n である．これは中央値や分位数を見つけるためのプログラム *find* の能力を示し，また候補全体をソートした後で k 番目を選ぶという単純な方法（この方法ではせいぜい $n \log n$ のオーダである）よりこの方法が優れていることを示している．しかし，最悪な場合には，各分割ステップにおい

て候補集合の大きさを1だけしか減らせないので，n^2 のオーダの比較回数が必要になる．さらに要素の数が少ない，たとえば，10以下のときにはこのアルゴリズムを使う利点はほとんどない．

2.2.8 配列のソーティング法の比較

各種ソーティング法の説明の締めくくりとして，それらの有効性の比較をしよう．n をソートすべき項目の総数とし，C と M は，それぞれ必要となるキー比較と項目移動の回数とする．3つの単純ソーティング法すべてに対して閉じた公式を導くことができるが，これは**表2.8**に与えられている．列の見出し最小，平均，最大は，それぞれ最小値，n 個の項目の $n!$ 個のすべての置換について平均した期待値，および最大値を表わす．

高度な方法については，程よく簡単で正確な公式は知られていない．しかし，必要な計算量が，Shell ソートでは $c_1 \cdot n^{1.2}$ であり，ヒープ・ソートやクィック・ソートでは $c_i \cdot n \cdot \log n$ であることは本質的な事実である．

これらの公式は，n の関数として性能を大ざっぱに測ったものであるが，これによってソーティング・アルゴリズムは，基本的または単純な方法 (n^2) と，高度なあるいは“対数的”方法 ($n \log n$) に類別される．しかし実際的な目的からは，係数 c_i の値をいくらかでも明らかにするような実験データを得ることは有用であり，それによって種々の方法をさらに類別することができる．その上これらの公式には，キー比較や項目移動以外の操作，たとえばループの制御等に費やされる計算が考慮されていない．明らかに，これらの因子は個々のシステムにある程度は依存するが，それでも，実験的に得られたデータ例は有益である．**表2.9**は，これまで述べてきた各種ソーティング法にかかる時間（ミリ秒）を示したものであるが，これは CDC 6600 計算機上の PASCAL シ

表2.8 単純ソーティング法の比較

		最小	平均	最大
単純挿入法	$C=$	$n-1$	$(n^2+n-2)/4$	$(n^2-n)/2-1$
	$M=$	$2(n-1)$	$(n^2-9n-10)/4$	$(n^2+3n-4)2$
単純選択法	$C=$	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
	$M=$	$3(n-1)$	$n(\ln n+0.57)$	$n^2/4+3(n-1)$
単純交換法	$C=$	$(n^2-n)/2$	$(n^2-n)/2$	$(n^2-n)/2$
（バブル・ソート）	$M=$	0	$(n^2-n)*0.75$	$(n^2-n)*1.5$

表2.9　各種ソート・プログラムの実行時間

	正順		ランダム		逆順	
単純挿入法	12	23	366	1444	704	2836
2分挿入法	56	125	373	1327	662	2490
単純選択法	489	1907	509	1956	695	2675
バブル・ソート	540	2165	1026	4054	1492	5931
フラグつきバブル・ソート	5	8	1104	4270	1645	6542
シェーカ・ソート	5	9	961	3642	1619	6520
Shell ソート	58	116	127	349	157	492
ヒープ・ソート	116	253	110	241	104	226
クイック・ソート	31	69	60	146	37	79
マージ・ソート†	99	234	102	242	99	232

† *2.3.1*参照

ステムで実行されたものである．3つの行は，既に正順になっている配列，ランダムな配列，および逆順の配列をソートするのに必要な時間を示している．各行の左側の数字は256個の項目また右側のものは512個の項目に対するものである．明らかに，このデータはn^2法と$n \log n$法を区別している．次の諸点に注目すべきである．

1. 2分挿入法の単純挿入法に対する改善度は限界に近く，既に正順になっている場合には，悪くさえなる．
2. バブル・ソートは明らかに比較したすべてのソーティング法のうちで最悪である．その改良版であるシェーカ・ソートでさえ，単純挿入法や単純選択法よりまだ悪い（既にソートされた配列をソートするような病理学的な場合を除いては）．
3. クィック・ソートはヒープ・ソートより2, 3倍良い．この方法は，逆順の配列を既にソートされた配列と実際的には同じ速さでソートする．

このデータは，キーだけから成り関連したデータを持たない項目をソートして得られたものであることを付け加えておかなければならない．これは，非常に現実的な仮定という訳ではない．**表2.10**は，項目のサイズが大きくなった場合の影響を示している．この例では，データはキーの7倍の記憶領域を占める．各列の左側の数字は，関連データのないレコードをソートするのに必要な時間を示し，右側の数字は，関連データを伴ったソーティングに関するものである．$n=256$である．

表2.10 ソート・プログラムの実行時間（キー＋関連データ）

	正	順	ランダム		逆	順
単純挿入法	12	46	366	1129	704	2150
2分挿入法	56	76	373	1105	662	2070
単純選択法	489	547	509	607	695	1430
バブル・ソート	540	610	1026	3212	1492	5599
フラグつき・バブルソート	5	5	1104	3237	1645	5762
シェーカ・ソート	5	5	961	3071	1619	5757
Shellソート	58	186	127	373	157	435
ヒープ・ソート	116	264	110	264	104	227
クィック・ソート	31	55	60	137	37	75
マージ・ソート†	99	196	102	195	99	187

† *2.3.1* 参照

次のような細目に注意してほしい：

1. 単純選択法はかなり良くなり，今回は単純な方法のうちで最良なものとなった.
2. バブル・ソートはまだ最悪な方法であり，他の方法との差が大きい（この方法はむしろ退歩した ）. そして“改良”された方法のうちでは，少しだけではあるが，シェーカ・ソートだけが逆順の場合にもとの方法より悪くなっている.
3. クィック・ソートは，最も高速な方法としての地位をさらに強め，配列のソーティング法としては最良であることが明らかとなった.

2.3 順ファイルのソーティング

2.3.1 単純マージ法

残念ながら，ソートすべきデータの量が多くて計算機の主記憶には格納できず，たとえばテープのような順次的周辺記憶装置上に置かれているときには，前節で与えたソーティング法を使うことはできない. この場合，データは(順)ファイルとして記述されるが，その特徴は一時にはただ1つの成分のみが直接アクセス可能なことである. これは，配列構造によって与えられる可能性と比べると厳しい制限であり，したがって異なったソーティング手法を使わなければならない. 最も重要な方法は，**マージ**†† によるソーティングである. マージ

†† 訳注 merge，併合ともいう.

とは，現在アクセス可能な成分の中から選択を繰り返すことにより2つ（またはそれ以上）の正順列を結合して1つの正順列を作ることを意味する．
マージはソートより簡単な操作であり，順次的ソーティングのより複雑なプロセス中で，補助的操作として使われる．マージに基づいたソーティングの一方法で**単純マージ法**と呼ばれる方法は，次のようなものである：

1. 列 a を半分に分割し，その各部分を b, c とする．
2. b, c から1つずつの項目を取り出し，それから正順対を作ることにより b, c をマージする．
3. マージされた列を a と呼び，ステップ1,2を繰り返す．この場合には正順対をマージして正順4組を作る．
4. 前のステップを繰り返して，4組をマージして8組を作り，…というようにこの操作を続けるが，そのたびにマージされる部分列の長さが2倍され，ついには，列全体がソートされる．

例として，次の列を考えよう．

44　55　12　42 | 94　18　06　67

ステップ1の分割により，次の2つの列が作られる．

44　55　12　42
94　18　06　67

1個の成分（これは，長さ1の正順列である）をマージして正順対を作る．

$44 \quad 94' \quad 18 \quad 55' \quad 06 \quad 12' \quad 42 \quad 67$

再び中央で分割し，正順対をマージすると，

$06 \quad 12 \quad 44 \quad 94' \quad 18 \quad 42 \quad 55 \quad 67$

3回目の分割とマージを行なうと，最後には求める結果が得られる．

06　12　18　42　44　55　67　94

データの集合全体を1回だけ扱う各操作は**フェイズ** (phase) と呼ばれ，またこの繰返しによりソート・プロセスを構成している最小の部分プロセスを**パス** (pass) または**ステージ** (stage) という．上の例では，ソートには3回のパスを要し，各パスは分割フェーズとマージ・フェーズから構成されている．このソートを実行するには3本のテープが必要であり，したがってこのプロセスは，

3テープ・マージと呼ばれる.

実際には，分割フェーズは項目の置換を全く行なわないので，ソートには寄与していない．それは，コピー操作の半分を占めているにもかかわらず，ある意味では非生産的である．分割とマージ・フェズを結合することによりこれを除去することは可能であるが，そのためには1本の列にマージする代わりに，マージ・プロセスの出力を2本のテープに直接分配し，それを次のパスの入力とすればよい．この方法は**1フェーズ・マージ**または**バランス・マージ**と呼ばれる．この方法は，半分のコピー操作しか必要としないという点で明らかに優れている．この利点のために支払われた代価は4番目のテープである．

これから，マージ・プログラムを詳細にわたって作り上げていくが，まず初めは，データが配列によって表現されているとする．ただし，この配列は厳密に順次的な形でしか走査されないと仮定する．その次に述べるマージ・ソート・プログラムは，ファイル構造に基づいたものにするつもりであるが，これによって，2つのプログラムの比較が可能となり，またプログラムの形が基礎となるデータ表現に依存することを示すことができる．

配列を両端からアクセス可能な列と考えることにより，2つのファイルの代わりに1つの配列を用いることは容易である．2つの入力ファイルからマージする代わりに，配列の両端から項目を取り出せばよい．このようにしたとき，結合されたマージ・分割フェーズの一般的な形は，**図2.12**のように示すことができる.
マージされた項目の行先は，最初のパスでは各正順対ごとに，また2番目のパスでは各正順4組ごとに，そして以下同様に切り換えられ，したがって，1つの配列の両端によって表わされる2つの出力列に均等に加えられる。各パスの後では，2つの配列はその役割を交換する．すなわち，入力は新しい出力に，

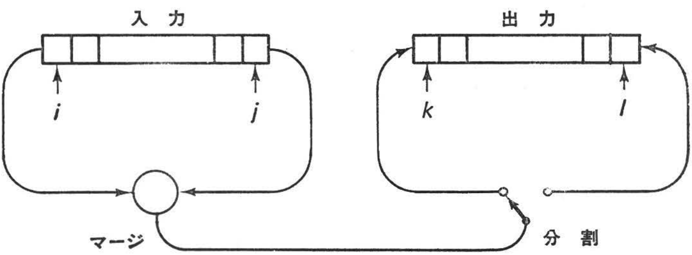


図2.12　2つの配列を用いた単純マージ・ソート

出力は新しい入力になる．

概念的に異なったこの2つの配列を結合して，大きさが2倍の1つの配列にすることにより，プログラムはさらに簡単になる．このとき，データは次のように表現される．

$$a : \textbf{array}\,[1\,..\,2*n]\ \textbf{of}\ item \tag{2.20}$$

そしてインデックス i と j は入力項目を，また k と l は出力項目を表わすとする（図2.12参照）．もちろん初期データは，$a_1 \cdots a_n$ である．明らかに，データの流れの方向を表わすために論理変数 up が必要となる：$up = true$ は，現在のパスで成分 $a_1 \cdots a_n$ が変数 $a_{n+1} \cdots a_{2n}$ のところに“上方向に”動移されることを意味し，一方，$up = false$ は $a_{n+1} \cdots a_{2n}$ が“下方向に”$a_1 \cdots a_n$ のところに移されることを意味する．相続くパスの間で up の値は厳密に交互に変わる．最後に変数 p を導入するが，これはマージする部分列の長さを表わすためである．その初期値は1であり，各パスの前に2倍される．プログラムを少し簡単化するために，n は常に2のべき乗と仮定する．このとき，単純マージ・プログラムの第1版は次のような形になる．

```
procedure mergesort;
    var i, j, k, l: index;
        up: boolean; p: integer;
begin up := true; p := 1;
   repeat {initialize indices}
      if up then
          begin i :=1 ; j := n; k := n+1; l :=2*n
          end else
          begin k := 1; l := n; i :=n+1; j :=2*n
          end;
      "i列とj列からのp組をマージして，k列とl列に出力する"
      up := ¬up; p := 2*p
   until p = n
end
```

次の展開ステップでは，言葉で表わした部分“i 列と j 列からの p 組をマージして，k 列と l 列に出力する”を詳細化する．明らかに，n 個の項目を含むこのマージ・パスは，それ自身，部分列である p 組のマージから成る列である．そのような各部分マージごとに，その行先は出力配列の下端から上端へ，

あるいは上端から下端へと切り換えられ，両方の行先への等しい分配が保証される．マージされた項目の行先が出力配列の下端の時には，行先インデックスは k であり，項目を1つ移動させるごとに k は1だけ増加される．もし，出力配列の上端に移動されるときには，行先インデックスは l であり，各移動ごとに l は1だけ減少される．実際のマージ文を簡単にするために，各 p 組のマージの後で変数 k と l の値を切り換え，行先はいつも k で表わすことにする．また，増分は常に h によって表わす．ただし，h は1または -1 である．プログラム設計に関するこれらの議論から次の詳細化が得られる．

```
h := 1; m:= n; {m = no.of items to be merged}
repeat q := p; r := p; m := m-2*p;
        "i からの q 個の項目と j からの r 個の項目をマージする.
        行先インデックスは k であり, 増分は h である";
        h := -h;
        "k と l を交換する"
until m = 0
```

次の詳細化ステップでは，実際のマージ文を定式化する．

このとき，マージ後どちらかの部分列に空でない部分が残されれば，この残りの部分は単純なコピー操作によって出力列に付加しなければならないことに注意しよう．

```
while (q≠0) ∧ (r≠0) do
begin {select an item from i or j}
   if a[i] .key < a[j] .key then
      begin "i から k に1つの項目を移し, i と k を進める"; q := q-1
      end else
      begin "j から k に1つの項目を移し, j と k を進める"; r := r-1
      end
end;
"i 列の残りをコピーする";
"j 例の残りをコピーする"                                   (2.23)
```

この後，残りの部分をコピーする操作をさらに詳細化すれば，プログラムは細部まで完全な形になる．その前に，n が2のべき乗であるという制限を取り

除きたい．この制約条件を緩めたときに，アルゴリズムのどの部分が影響を受けるであろうか．より一般的な状況に対処する最良の方法は，可能な限り現在の方法に固執することである，ということは容易に納得できる．この例では，これは入力列の残りが長さ p 以下になるまで p 組のマージを続けることを意味する．影響を受ける1つの，そして，ただ1つの部分は，q と r の値，すなわち，マージされる列の長さを決定する部分である．3つの文

$$q:=p;\ r:=p;\ m:=m-2*p$$

は次の4つの文で置き換えられるが，これが上に述べた方針を有効に適用したものであることは，読者も納得されるであろう．m は，2つの入力列中の項目で，まだマージしていないものの総数を表わすことに注意しよう．

$$\textbf{if } m \geq p \textbf{ then } q:=p \textbf{ else } q:=m;\ m:=m-q;$$
$$\textbf{if } m \geq p \textbf{ then } r:=p \textbf{ else } r:=m;\ m:=m-r;$$

さらにプログラムの停止を保証するために，"外側"の繰り返しを制御している条件 $p=n$ は，$p \geq n$ に変えなければならない．これらの修正によって，いよいよ，アルゴリズム全体を完全なプログラムの形に記述することができる（**プログラム 2.13** 参照）．

マージ・ソートの解析　各パスで p は2倍され，また，$p \geq n$ になり次第ソートは停止されるので，$\lceil \log_2 n \rceil$ 回のパスが実行される．定義より，各パスは n 個の項目からなる集合をちょうど1回だけコピーする．その結果，移動の総回数は，

$$M = n \cdot \lceil \log n \rceil \tag{2.24}$$

である．比較の回数 C は M よりもっと少ない．なぜなら，残りの部分のコピー操作には比較が含まれないからである．しかしながら，マージ・ソート技法は周辺記憶装置の使用と関連して用いられるのが普通であるので，移動操作に必要な計算時間は，比較に必要な時間に比べてその大きさが何桁か大きい．したがって，比較回数の正確な解析は実用上ほとんど興味がない．

明らかに，マージ・ソート・アルゴリズムは，前節で論じた高度なソーティング技法と比較しても同等な性能をもっている．しかし，インデックス操作の管理のためのオーバヘッドが比較的大きく，また，$2n$ 個の項目のための記憶

プログラム 2.13 単純マージ・ソート

```
procedure mergesort;
   var i,j,k,l,t: index;
      h,m,p,q,r: integer; up: boolean;
   {note that a has indices 1 . . . 2*n}
begin up := true; p := 1;
   repeat h := 1; m := n;
      if up then
      begin i := 1; j := n; k := n+1; l := 2*n
      end else
      begin k := 1; l := n; i := n+1; j := 2*n
      end ;
      repeat {merge a run from i and j to k}
         {q = length of i-run, r = length of j-run}
         if m ≥ p then q := p else q := m; m := m−q;
         if m ≥ p then r := p else r := m; m := m−r;
         while (q≠0) ∧ (r≠0) do
         begin {merge}
            if a[i] .key < a[j] .key then
            begin a[k] := a[i]; k := k+h; i := i+1; q := q−1
            end else
            begin a[k] := a[j]; k := k+h; j := j−1; r := r−1
            end
         end ;
         {copy tail of j-run}
         while r ≠ 0 do
            begin a[k] := a[j]; k := k+h; j := j−1; r := r−1
            end ;
         {copy tail of i-run}
         while q ≠ 0 do
            begin a[k] := a[i]; k := k+h; i := i+1; q := q−1
            end ;
         h := −h; t := k; k := l; l := t
      until m = 0;
      up := ¬up; p := 2*p
   until p ≥ n;
   if ¬up then
   for i := 1 to n do a[i] := a[i+n]
end {mergesort}
```

領域が必要なことは決定的な弱点である．これが，配列すなわち主記憶中のデータに対してマージによるソーティングがほとんど使われない理由である．このマージ・ソート・アルゴリズムの実時間的動作を比較するための数字が表2.9と2.10の最後の行に与えられている．それらはヒープ・ソートより良いが，クィック・トソートより悪い．

2.3.2　自然なマージ

単純マージでは，データが初めに部分的にソートされていても何の利益も得られない．k 番目のパスでマージされる部分列の長さは全て 2^k に等しく（または短かく），マージ可能なより長い正順な部分列が存在しているか否かには無関係である．

実際には，長さ m と n の正順な部分列は，$m+n$ の項目からなる列に直接にマージすることができる．常に最も長い部分列をマージするマージ・ソートは，**自然なマージ・ソート**と呼ばれる．

正順な部分列は，しばしば**ストリング** (string) と呼ばれる．しかしストリングという語は，文字列を表わすのに使われることの方が多いので，われわれはKnuth にしたがって，正順な部分列を指すのに，語 **連** (run) をストリングの代わりに用いる†．次を満たす部分列 $a_i \cdots a_j$ を**極大な連**，または単に連という．

$$
\begin{aligned}
&a_k \le a_{k+1} \quad \text{ただし} \quad k = i, \cdots, j-1 \\
&a_{i-1} > a_i \\
&a_j > a_{j+1}
\end{aligned}
\qquad (2.25)
$$

したがって，自然なマージ・ソートでは，一定の決められた長さの部分列の代わりに（極大な）連をマージする．n 個の連から成る2つの列をマージすると，ちょうど n 個の連から成る1つの列ができる，という性質を連はもっている．よって，各パスごとに連の総数は半分になり，ゆえに必要となる項目の移動回数は，最悪の場合でも $n \cdot \lceil \log_2 n \rceil$ であり，また平均的にはさらに少ない．しかし，項目の選択に必要な比較のほかに，各連の端を決定するために各ファイルの連続した項目間の比較を行なうことがさらに必要になるので，比較回数の期待値はもっと大きい．

次のプログラミングの問題として，自然なマージ・アルゴリズムを構成する

† **訳注**　統計学で run というときには，同一の文字の連続をいい，このときには連という日本語が使われている．ここでの run の意味は少し異なるが，項目間で成立する≦，≧について考えれば上記の意味での run になっており，したがって run は連と訳してある．

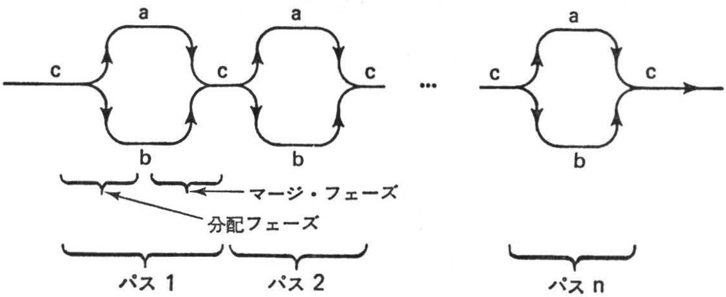


図 2.13 ソート・フェーズとソート・パス

表 2.11 自然なマージ・ソートの例

17	31′	5	59′	13	41	43	67′	11	23	29	47′	3	7	71′	2	19	57′	37	61
5	17	31	59′	11	13	23	29	41	43	47	67′	2	3	7	19	57	71′	37	61
5	11	13	17	23	29	31	41	43	47	59	67′	2	3	7	19	37	57	61	71
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	57	59	61	67	71

が，それには，単純マージ・アルゴリズムの説明に用いたのと同じような段階的方法を用いる．今度は，配列の代わりに順ファイル構造を採用し，またバランスしていない2フェーズ・3テープ・マージ・ソートを実現する．項目の初期列はファイル c によって与えられ，またソートされた結果もその上に出力すると仮定する（実際のデータ処理への応用では，安全のためにまず最初に初期データは原テープからファイル c 上にコピーされるのが普通である）．2本の補助テープを a，b とする．各パスは，c から a と b 上に連を平等に分配する分配フェーズと，a と b から連を c 上にマージするマージ・フェーズから構成される．このプロセスは，**図 2.13** に示されている．

1例として，20個の数に対する自然なマージ・ソートが**表 2.11** に与えられているが，初期状態（1行目）と各パス後（2-4行目）のファイル c の内容が示されている．ソートは c 上の連の数が1になったときに停止する（初期ファイル上には，少なくとも1つの空でない連が存在すると仮定する）．したがって，ファイル c 上にマージされた連の個数を数えるために1つの変数 l を用いる．大域的変数を

```
type tape = file of item;
var c: tape
```
(2.26)

のように定義すると，プログラムは次のように定式化される．

```
procedure naturalmerge;
    var l: integer;
        a,b: tape;
begin
    repeat rewrite(a); rewrite(b); reset(c);
        distribute;                                         (2.27)
        reset(a); reset(b); rewrite(c);
        l := 0; merge
    until l = 1
end
```

明らかに，2つのフェーズは2つの異なった文として表現されている．これからそれらを詳細化する．詳細化された記述は，直接に代入されるか，あるいは，手続きとして記述され，省略形の文を手続き呼出しと考える．今回は後者の方法を用いることにし，次のように定義する．

```
procedure distribute; · {from c to a and b}
begin
    repeat copyrun(c,a);                                    (2.28)
        if ¬eof(c) then copyrun(c,b)
    until eof(c)
end
```

および

```
procedure merge;
begin   {from a and b to c}
    repeat mergerun; l := l+1
    until eof(b);                                           (2.29)
    if ¬eof(a) then
        begin copyrun(a,c); l := l+1
        end
end
```

この分配方法によって，ファイル a と b 上に同数の連が作り出されるか，あるいはファイル a には b よりも1個だけ多くの連が含まれる，と推定される．対応する連の対はマージされるので，ファイル a 上には1つの連が残される可能性があり，これは c 上にコピーされなければならない．手続き *merge* と

distribute は，内容の明らかな下位手続き *mergerun* と *copyrun* を使って定義されている．以後，これらの手続きをさらに詳細に説明する．そのためには，連の終わり (*end of run*) に達したかどうかを指定するための大域的論理変数 *eor* を導入することが必要である．

```
procedure copyrun(var x,y: tape);
begin  {copy one run from x to y}                          (2.30)
    repeat copy(x,y) until eor
end
```

```
procedure mergerun;
begin  {merge a run from a and b to c}
    repeat if a↑.key < b↑.key then
        begin copy(a,c);
            if eor then copyrun(b,c)
        end else
        begin copy(b,c);                                   (2.31)
            if eor then copyrun(a,c)
        end
    until eor
end
```

連をマージするとき，キーの比較と選択のプロセスは，2つの連のいずれかが空になったときに停止する．この後，（まだ空になっていない）もう一方の連は，その残りの部分を単にコピーすることにより結果の連に移される．これは手続き *copyrun* の呼び出しによって行なわれる．

この2つの手続きは，下位手続き *copy* を使って定義されているが，これは入力ファイル x から出力ファイル y に1つの項目を転送し，そして連の終わりに達したかどうかを決定する．この手続きは，*read* 文と *write* 文を使って容易に表わせる．連の終わりを決定するためには，最後に読まれた（コピーされた）項目を記憶しておき，次のものと比較すればよい．この“先読み”は，ファイルのバッファ変数 $x↑$ を調べることにより実行される．

```
procedure copy(var x,y: tape);
    var buf: item;
begin read(x, buf); write(y, buf);                          (2.32)
    if eof(x) then eor := true else eor := buf.key > x↑.key
end
```

以上によって，自然なマージ・ソート・アルゴリズムの作成が終了した．しかし，注意深い読者なら気がついたであろうが，残念ながらこのプログラムは正しくない．すなわち，正しくソートしない場合がある．たとえば，次の入力データの例を考えよう．

3　2　5　11　7　13　19　17　23　31　29　37　43　41　47　59　57
61　71　67

相続く連をファイル a と b 上に交互に分配することにより，

$$a = 3'\ \ 7\ \ 13\ \ 19'\ \ 29\ \ 37\ \ 43'\ \ 57\ \ 61\ \ 71$$
$$b = 2\ \ 5\ \ 11'\ \ 17\ \ 23\ \ 31'\ \ 41\ \ 47\ \ 59'\ \ 67$$

これらの列はただちにマージされて1つの連になり，ソートは成功する．プログラムは間違った動作をしなかったが，連をいくつかのファイル上にただ分配するだけでは，出力ファイル中の連の数が入力ファイル中の連の数より少なくなる可能性があることを，この例は示している．これは，$i+2$番目の連の最初の項目がi番目の連の最後の項目より大きい場合には，2つの連が自動的に1つの連にマージしてしまうからである．

したがって，手続き *distribute* は2つのファイル上に等しく連を出力すると考えたが，a と b 上に作り出される連の実際の個数は，かなり違っている可能性がある．しかし，われわれのマージ手続きは連の対をマージするだけであり，ファイル b を読み終わると停止してしまうので，一方のファイルの残りの部分が失われてしまう．**表2.12**の入力データを考えてみよう．これは，2回のパスでソートされる（そして切り捨てられる）．

このプログラムの誤りは，プログラミング上よくある典形的なものである．この誤りは，簡単だと思われる操作の結果について，その可能性の1つを見落としたことによるものである．この誤りをなおすにはいくつかの方法があるが，そのうちのどれかを選択しなければならないという意味でもまた典形的である．このような場合，非常に重要かつ基本的な点で異なる2つの可能性のあることが多い：

表2.12　マージ・ソート・プログラムの正しくない結果

17	19	13	57	23	29	11	59	31	37	7	61	41	43	5	67	47	71	2	3
13	17	19	23	29	31	37	41	43	47	57	71	11	59						
11	13	17	19	23	29	31	37	41	43	47	57	59	71						

1. 分配の操作が間違ってプログラムされており，連の数が等しい（またはたかだか1つしか違わない）という要求を満たしていないことをわれわれは認める．われわれは，この操作のオリジナルな形式を守り，それに従って間違っている手続きを直す．
2. 間違っている部分の訂正によって，さらに別の部分の修正が必要となることがわかる．したがって，アルゴリズムの他の部分を変更して，現在誤っている部分を吸収する方法を見つけるように努力する．

一般に，最初の方法の方が安全であり，また，明解かつ直接的であるので，それによる複雑な副作用を見落とすという結果をかなり避けることのできる方法のように思われる．したがって，これは一般的に（そして正当に）推奨できる方法である．

しかし，ときには2番目の可能性も完全には無視できないことに注意すべきである．ここでは，本来間違っている分配手続きではなく，マージ手続きを修正することにより，この例題をさらに練りなおすことにする．すなわち，分配の方法には手をつけずに，連が等しく分配されるという条件を放棄する．これにより，最適な場合より多少性能は落ちるが，最悪な場合の性能は変わらない．そのうえ，非常に不つりあいな分配は，統計的には**ごくまれ**にしか起こらない．したがって，効率の面でこの解が特に劣るということはない．

連の等しい分配という条件が存在しないとすれば，マージ手続きは次のように変更しなければならない．すなわち，1つのファイルの終わりに達したならば，もう1つのファイルは，たかだか1つの連だけではなく，残っている部分**全体**をコピーする．

この変更は簡単であり，分配の方法を変えるどのような方法と比較しても非常に単純である（読者はぜひこの主張の正しいことを納得してほしい）．マージ・アルゴリズムの改良版は，完結した**プログラム 2.14** に含まれている．

プログラム 2.14 自然なマージ・ソート

```
program mergesort (input, output);
{3-tape, 2-phase natural merge sort}
type item = record key: integer
                {other fields defined here}
            end ;
   tape = file of item;
var c: tape; n: integer; buf: item;
```

```
procedure list (var f: tape);
   var x: item;
begin reset(f);
   while ¬eof(f) do
      begin read(f,x); write(output, x.key)
      end ;
   writeln
end {list} ;

procedure naturalmerge;
   var l: integer;   {no. of runs merged}
      eor: boolean; {end -of -run indicator}
      a,b: tape;
   procedure copy(var x,y: tape);
      var buf: item;
   begin read(x, buf); write(y,buf);
      if eof(x) then eor := true else eor := buf.key > x↑.key
   end ;

   procedure copyrun (var x,y: tape);
   begin {copy one run from x to y}
      repeat copy(x,y) until eor
   end ;
   procedure distribute;
   begin {from c to a and b}
      repeat copyrun (c,a);
         if ¬eof(c) then copyrun (c,b)
      until eof(c)
   end ;
   procedure mergerun;
   begin {from a and b to c}
      repeat
         if a↑.key ≤ b↑.key then
         begin copy (a,c);
            if eor then copyrun (b,c)
         end else
         begin copy (b,c);
            if eor then copyrun (a,c)
         end
      until eor
   end ;
```

```
    procedure merge;
    begin {from a and b to c}
        while ¬eof(a) ∧ ¬eof(b) do
        begin mergerun; l := l+1
        end;
        while ¬eof(a) do
        begin copyrun (a,c); l := l+1
        end;
        while ¬eof(b) do
        begin copyrun (b,c); l := l+1
        end ;
        list (c)
    end ;
begin
    repeat rewrite(a); rewrite(b); reset(c);
        distribute;
        reset(a); reset(b); rewrite(c);
        l := 0; merge;
    until l = 1
end ;
begin {main program; read input sequence ending with 0}
    rewrite(c); read(buf.key);
    repeat write(c.buf); read(buf.key)
    until buf .key = 0;
    list (c);
    naturalmerge;
    list(c)
end
```

2.3.3 バランス・マルチウェイ・マージ

各パスでは全データのコピーが行なわれるので，順次的ソートに必要な計算量は，必要となるパスの回数に比例する．この回数を減らす1つの方法は，連を2つ以上のファイルに分配することである．N本のテープ上に等しく分配されているr個の連をマージすれば，r/N個の連が生じる．2回目のパスの後ではその数はr/N^2に，3回目ではr/N^3となり，そしてk回のパスの後では，r/N^k個の連が残されることになる．したがって，Nウェイ・マージによってn個の項目をソートするのに必要なパスの全回数は，$k=\lceil \log_N n \rceil$である．各

パスでは n 回のコピー操作が必要なので，コピー操作の全回数は，最悪な場合には次のようになる．

$$M = n \cdot \lceil \log_N n \rceil$$

次のプログラミングの問題として，マルチウェイ・マージに基づいたソート・プログラムを作るが，前の自然な2フェーズ・マージ手続きと対比するために，1フェーズ・バランス・マージ・ソートとしてこのマルチウエイ・マージを定式化する．これは，各パスで同数の入力と出力ファイルがあり，相続く連は出力ファイル上に交互に分配されることを意味する．したがって，Nが偶数の時には，N 個のファイルを使うこのアルゴリズムは，$N/2$ ウエイ・マージに基づくものである．前に採用された方針に従って，同じテープの上に分配される2つの相続く連が自動的にマージしてしまうことの検出は行なわない．したがって，各入力テープ上には厳密に等しい数の連が存在するということを仮定せずに，マージ・プログラムを設計しなければならない．

このプログラムにおいて，われわれは，ファイルから成る配列というデータ構造の自然な応用にはじめて出会う．それにしても2ウエイからマルチ・ウエイに変更したことにより，このプログラムが前のものとは非常に異なったものになったということは驚きである．この変化は，主に，入力連のどれかが空になったという条件だけではマージ・プロセスをもはや停止することができない，という事情によるものである．その代わりに，現在まだ生きている，すなわち空になっていない入力のリストを保持していなければならない．各パスごとに入力テープと出力テープを切り換えなければならないことも，プログラムを複雑にさせる原因のひとつである．

今までよく使ってきた型 *item* と *tape* に加えて，まず次の型を導入する．

$$tapeno = 1 \,..\, n \tag{2.33}$$

明らかに，*tapeno*（テープ番号）は項目ファイルからなる配列のインデックに用いられる．次に，初期項目列は変数 $f0$ によって与えられると仮定する．

$$f0 : tape \tag{2.34}$$

また，ソーティングのために n 本のテープが利用できるとする．

$$f : \textbf{array}\ [tapeno]\ \textbf{of}\ tape \tag{2.35}$$

ただし，n は偶数とする．

テープ切換えの手法としては，テープ・インデックス写像を利用するのがよい．インデックスによってテープを直接に指定する代わりに，写像 t を介して指定する．すなわち，

$$f[i] \text{の代わりに} f[t[i]]$$

を用いる．ここで，この写像は

$$t\text{: }\mathbf{array}\ [tapeno]\ \mathbf{of}\ tapeno \tag{2.36}$$

のように定義される．初めは全ての i に対して $t[i]=i$ であり，テープ切換えには，単に写像成分の対を交換すればよい．

$$\begin{aligned} t[1] &\longleftrightarrow t[nh+1] \\ t[2] &\longleftrightarrow t[nh+2] \\ &\cdots \\ t[nh] &\longleftrightarrow t[n] \end{aligned}$$

ここで，$nh=n/2$ である．したがって，われわれは常に

$$f[t[1]],\ \cdots,\ f[t[nh]]$$

を入力テープと考え，

$$f[t[nh+1]],\ \cdots,\ f[t[n]]$$

を出力テープと考える．（以後，コメントの中では $f[t[j]]$ を単に "*tape j*" と書く）．以上の準備のもとに，アルゴリズムは初め次のように定式化される．

```
procedure tapemergesort;
    var i,j: tapeno;
        l: integer; {no. of runs distributed}
        t: array [tapeno] of tapeno;
begin {distribute initial runs to t[1] . . . t[nh]}
    j := nh; l := 0;
    repeat if j < nh then j := j+1 else j := 1;
            "copy one run from f0 to tape j";
            l := l+1
    until eof(f0);
```

(2.37)

```
    for i := 1 to n do t[i] := i;
    repeat {merge from t[1] ... t[nh] to t[nh+1] ... t[n]}
           "reset input tapes";
           l := 0;
           j := nh+1; {j = index of output tape}
           repeat l := l+1;
                  "merge a run from inputs to t[j]"
              if j < n then j := j+1 else j := nh+1
           until "all inputs exhausted";
           "switch tapes"
    until l = 1;
    {sorted tape is t[1]}
end
```

次に，連の初期分配に使われるコピー操作を詳細化する．再び，最後に読んだ項目を保持するための補助変数を導入する．

$$buf : item$$

そして，"*copy one run from f0 to tape j*"（1つの連を $f0$ からテープ j にコピーする）を次の文で置き換える．

```
repeat read(f0, buf);
    write(f[j], buf)                              (2.38)
until (buf.key > f0↑.key) ∨ eof(f0)
```

1つの連のコピーは，次の連の最初の項目が見つかる ($buf.key > f0\uparrow.key$) か，入力ファイルの終わりになったとき ($eof(f0)$) に停止する．

実際のプログラムでは，まだ次の文

1. "*reset input tapes*"（入力テープをリセットする）
2. "*merge a run from inputs to t[j]*"（入力テープから1つの連を $t[j]$ にマージする）
3. "*switch tapes*"（テープを切り換える）

と述語

4. "*all inputs exhausted*"（すべての入力テープが空になった）

をもっと詳細に指定する必要がある．まず初めに，現在の入力テープを正確に

同定しなければならない．明らかに“生きている”入力ファイルの数は $n/2$ より小さい．実際には，たかだか連の数と同じだけしかそのような入力ファイルは存在し得ず，そして1つのファイルだけになったときにソートは終了する．このことは，最後のソート・パスの初めには，nh 個以下の連が存在する可能性を残している．したがって，使用される入力ファイルの実際の数を表わすために，変数 $k1$ を導入する．$k1$ の初期設定は文“*reset input tapes*”の中に次のように組み込む：

if $l < nh$ **then** $k1 := l$ **else** $k1 := nh$;
for $i := 1$ **to** $k1$ **do** *reset*($f[t[i]]$);

もちろん文(2)では，ある入力ファイルが空になったとき $k1$ を減らす．したがって，述語(4)は次の関係によって容易に表わせる．

$$k1 = 0$$

文(2)を詳細化するのはもっと困難である．それは，現在対象としているキーの中から最小のものを選び，それをその行先，すなわち現在の出力テープに転送することを繰り返すことから成り立っている．このプロセスは，連の終わりを決定する必要性によりさらに複雑になっている．連の終わりに達するのは，（1）次のキーが現在のキーより小さいか，または（2）入力ファイルの終わりに達したときである．後者の場合には，$k1$ を減じることによりそのテープを消去する．前者の場合には，その連を閉じる．そして，出力連の構成が完了するまで項目の選出対象からこのファイルを除外する．このことから，次の項目の選択に実際に使われる入力テープの本数を表わすために2番目の変数，たとえば $k2$，が必要であることは明らかである．この値は $k1$ に初期設定され，条件（1）により連が終端するごとに減少させられる．

残念なことに $k2$ の導入だけでは十分ではない．すなわち，テープの本数について知っているだけでは十分ではなく，現在まだ残っているテープがどれであるかを知っている必要がある．このための明らかな解は，論理型の成分をもった配列を用いて，テープの使用可能性を示すことである．しかし，われわれは別の方法によって，もっと効率の良い選択アルゴリズムを構成する．なぜなら，これはアルゴリズム全体の中で最も頻繁に用いられる部分だからである．論理配列を用いる代わりに2番目の写像，たとえば ta，を導入する．この写像は t の代わりに用いられ，$ta[1] \cdots ta[k2]$ は使用可能なテープのインデックスを表わす．文（2）は次のように定式化できる：

```
k2 := k1;
repeat "select the minimal key, let ta[mx] be its tape number";
        read(f[ta[mx]], buf);
        write(f[t[j]], buf);                                    (2.39)
        if eof(f[ta[mx]]) then "eliminate tape" else
        if buf.key > f[ta[mx]]↑.key then "close run"
until k2 = 0
```

どの計算機システムでも，実際に使用可能なテープ装置の数はそれほど多くないので，次の詳細化ステップで記述される選択アルゴリズムには，単純な線形探索を使うことができる．文 "*eliminate tape*"（テープを消去する）は，*k*1 と *k*2 を減らし，写像 *ta* におけるインデックスの再割当てを行なうことを意味する．文 "*close run*"（連を閉じる）は，単に *k*2 を減らし，それに従って *ta* の成分の再配置を行なう．その詳細は**プログラム 2.15** に示されているが，このプログラムは，(2.37) から (2.39) の詳細化の最後のものである．テープは，その最後の連が読まれた直後に，手続き *rewrite* によって巻き戻されていることに注意せよ．文 "*switch tapes*"（テープを切換える）は，前に述べた説明にしたがって詳細化されている．

プログラム 2.15　バランス・マージ・ソート

```
program balancedmerge (output);
{balanced n-way tape merge sort}
const n = 6; nh = 3;          {no. of tapes}
type item = record
                key: integer
            end ;
   tape = file of item;
   tapeno = 1 .. n;
var leng, rand: integer;      {used to generate file}
   eot: boolean;              {end of tape}
   buf: item;
   f0: tape; {f0 is the input tape with random numbers}
   f: array [1 .. n] of tape;
```

```
procedure list(var f: tape; n: tapeno);
   var z: integer;
begin writeln('TAPE', n:2); z := 0;
   while ¬eof(f) do
   begin read(f, buf); write(output, buf.key: 5); z := z+1;
      if z = 25 then
         begin writeln(output); z := 0;
         end
   end ;
   if z ≠ 0 then writeln (output); reset(f)
end {list} ;

procedure tapemergesort;
   var i,j,mx,tx: tapeno;
      k1,k2,l: integer;
      x, min: integer;
      t, ta: array [tapeno] of tapeno;
begin {distribute initial runs to t[1] . . . t[nh]}
   for i := 1 to nh do rewrite(f[i]);
   j := nh; l := 0;
   repeat if j < nh then j := j+1 else j := 1;
      {copy one run from f0 to tape j}
      l := l+1;
      repeat read(f0, buf); write(f[j], buf)
      until (buf .key > f0↑ .key) ∨ eof(f0)
   until eof(f0);
   for i := 1 to n do t[i] := i;
   repeat {merge from t[1] . . . t[nh] to t[nh+1] . . . t[n]}
      if l < nh then k1 := l else k1 := nh;
      {k1 = no. of input tapes in this phase}
      for i := 1 to k1 do
         begin reset(f[t[i]]); list(f[t[i]], t[i]); ta[i] := t[i]
         end ;
      l := 0;  {l = number of runs merged}
      j := nh+1;      {j = index of output tape}
      repeat {merge a run from t[1] . . . t[k1] to t[j]}
         k2 := k1; l := l+1;  {k2 = no. of active input tapes}
         repeat {select minimal element}
            i := 1; mx := 1; min := f[ta[1]]↑.key;
            while i < k2 do
```

```
            begin i := i+1; x := f[ta[i]]↑.key;
                if x < min then
                begin min := x; mx := i
                end
            end ;
            {ta[mx] has minimal element, move it to t[j]}
            read(f[ta[mx]], buf); eot := eof(f[ta[mx]]);
            write(f[t[j]], buf);
            if eot then
            begin rewrite(f[ta[mx]]); {eliminate tape}
                ta[mx] := ta[k2]; ta[k2] := ta[k1];
                k1 := k1-1; k2 := k2-1
            end else
            if buf .key > f[ta[mx]]↑ .key then
            begin tx := ta[mx]; ta[mx] := ta[k2]; ta[k2] := tx;
                k2 := k2-1
            end
        until k2 = 0;
        if j < n then j := j+1 else j := nh+1
    until k1 = 0;
    for i := 1 to nh do
        begin tx := t[i]; t[i] := t[i+nh]; t[i+nh] := tx
        end
  until l = 1;
  reset(f[t[1]]); list(f[t[1]], t[1]);  {sorted output is on t[1]}
end {tapemergesort} ;

begin {generate random file f0}
  leng := 200; rand := 7789; rewrite(f0);
  repeat rand := (131071*rand) mod 2147483647;
    buf .key := rand div 2147484; write(f0, buf); leng := leng - 1
  until leng = 0;
  reset(f0); list(f0, 1);
  tapemergesort
end
```

2.3.4　ポリフェーズ・ソート

ここではバランス・ソートより性能の優れたもう１つのソーティング・アルゴリズムについて調べそのプログラムを作るが，そのために必要な技法や基礎知識はすでに習得してきた．バランス・マージでは，分配とマージ操作を１つ

のフェーズに結合することにより，純粋なコピー操作は必要でないことがわかった．次には，与えられたテープをもっと効率よく利用できないかという疑問が生じるが全くその通りである；次の改良に対する鍵は，厳密なパスという概念を捨てることである．すなわち，常に $N/2$ 本ずつの入力および出力テープを用意し，それらを各パスの終わりで交換するという方法より更に凝った方法でテープを使用することである．この方法は，R. L. Gilstad[2-3] により考案され，**ポリフェーズ・ソート** (polyphase sort) と命名されたが，この方法ではパスの概念はあいまいである．

まず，3本のテープを使った例によってこの方法を説明しよう．常に，項目は2本のテープから3番目のテープ上にマージされる．入力テープのうちの1本が空になると，ただちにそれは出力テープになり，まだ空になっていないテープと前の出力テープを新しい入力テープとするマージが行なわれる．

各入力テープ上の n 個の連は，出力テープ上の n 個の連に変換されることがわかっているので，（実際のキーを指定する代わりに）各テープ上に存在している連の数だけを書くだけで十分である．**図 2.14** では，初めに入力テープ $f1$ と $f2$ は，それぞれ13個と8個の連を含んでいたと仮定する．したがって，最初の“パス”では，8個の連が $f1$ と $f2$ から $f3$ 上にマージされ，2番目の“パス”では残っている5個の連が $f3$ と $f1$ から $f2$ 上にマージされる，というようにマージが行なわれる．そして最後には $f1$ がソートされたファイルと

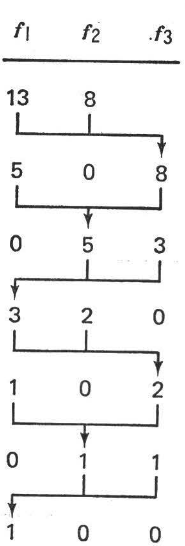


図 2.14 3本のテープを用いた21個の連のポリフェーズ・マージ・ソート

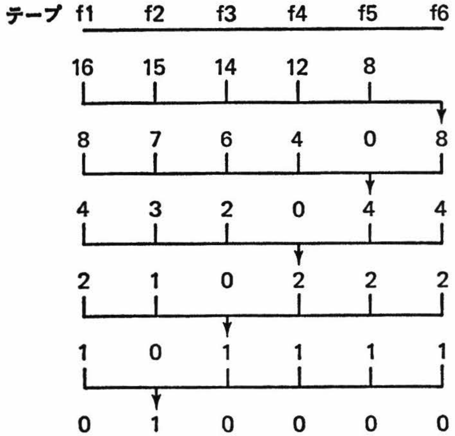


図 2.15　6本のテープによる65個の連のポリフェーズ・マージ・ソート

なる．

次の例では，6テープを用いたポリフェーズ法を示す．最初に，$f1$ 上に 16，$f2$ 上に 15，$f3$ 上に 14，$f4$ 上に 12，$f5$ 上に 8 個の連があったとする．最初の部分パスでは，8個の連が $f6$ 上にマージされ，最後には，$f2$ がソートされた項目の集合を含んでいる（図 2.15 参照）．

ポリフェーズ法はバランス・マージ法より効率がよい．なぜなら，N本のテープが与えられたとき，$N/2$ ウエイ・マージではなく，常に $N-1$ ウエイ・マージで動作するからである．n をソートすべき項目の数，N をマージ操作の次数† とすると，必要となるパスの回数は近似的に $\log_N n$ であるので，ポリフェーズ法はバランス・マージ法に対する重要な改良になっていることが期待さ

表 2.13　2テープ上への連の完全な分配

l	$a_1^{(l)}$	$a_2^{(l)}$	$\Sigma a_i^{(l)}$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

† 訳注　入力テープの本数

表 2.14　5テープ上への連の完全な分配

l	$a_1^{(l)}$	$a_2^{(l)}$	$a_3^{(l)}$	$a_4^{(l)}$	$a_5^{(l)}$	$\Sigma a_i^{(l)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

れる．

もちろん，上の例題では連の初期分配は注意深く行なわれていた．正しく動作するための連の初期分配を見つけるためには，最後の分配（図 2.15 の最後の行）から出発して逆向きに調べればよい．2つの例題の表を書き直し，各行を前の行に関して1つだけ位置を回転させることにより，3および6テープのそれぞれについて**表 2.13**と**表 2.14**が得られる．ただし，$a_i^{(l)}$ は終わりから数えて l 番目の部分パス後の各テープ上の連の数を表わす（訳者挿入）．

表 2.13 から $l \geq 0$ に対して次の関係が導かれる．

$$\begin{aligned} a_2^{(l+1)} &= a_1^{(l)} \\ a_1^{(l+1)} &= a_1^{(l)} + a_2^{(l)} \end{aligned} \tag{2.40}$$

また $a_1^{(0)}=1, a_2^{(0)}=0$ である．$a_1^{(i)} = f_{i+1}$ とおくことにより

$$\begin{aligned} f_{i+1} &= f_i + f_{i-1}, \qquad ただし\ i \geq 1 \\ f_1 &= 1 \\ f_0 &= 0 \end{aligned} \tag{2.41}$$

これらは，いわゆる**フィボナッチ** (Fibonacci) 数

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \cdots$$

を定義する再帰規則（または再帰関係）である．各フィボナッチ数は，その直前の2つの数の和である．したがって，3テープによるポリフェーズ法が正しく動作するためには，2本のテープ上の初期連の数は相続くフィボナッチ数でなければならない．

6テープの例（表 2.14）の場合はどうであろうか．次の規則が容易に導かれる．

$$
\begin{aligned}
a_5^{(l+1)} &= a_1^{(l)} \\
a_4^{(l+1)} &= a_1^{(l)} + a_5^{(l)} = a_1^{(l)} + a_1^{(l-1)} \\
a_3^{(l+1)} &= a_1^{(l)} + a_4^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} \\
a_2^{(l+1)} &= a_1^{(l)} + a_3^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} + a_1^{(l-3)} \\
a_1^{(l+1)} &= a_1^{(l)} + a_2^{(l)} = a_1^{(l)} + a_1^{(l-1)} + a_1^{(l-2)} + a_1^{(l-3)} + a_1^{(l-4)}
\end{aligned}
\tag{2.42}
$$

f_{i+4} を $a_1^{(i)}$ に代入すると，

$$
\begin{aligned}
& f_{i+1} = f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4}, \qquad \text{ただし } i \geq 4 \\
& f_4 = 1 \\
& f_i = 0 \qquad \text{ただし } 0 \leq i < 4
\end{aligned}
\tag{2.43}
$$

こらの数は次数4のフィボナッチ数といわれる．一般に，**次数 p のフィボナッチ数**は次のように定義される．

$$
\begin{aligned}
& f_{i+1}^{(p)} = f_i^{(p)} + f_{i-1}^{(p)} + \cdots + f_{i-p}^{(p)}, \quad \text{ただし } i \geq p \\
& f_p^{(p)} = 1 \\
& f_i^{(p)} = 0, \quad \text{ただし } 0 \leq i < p
\end{aligned}
\tag{2.44}
$$

通常のフィボナッチ数は，次数1のものであることに注意せよ．

以上より，n 本のテープを使った完全なポリフェーズ・ソートに対しては，連の初期数は連続した $n-1$, $n-2$, …，1個の次数 $n-2$ のフィボナッチ数の和であることがわかる（**表 2.15** 参照）．
したがって，明らかにこの方法が適用可能なのは，連の数がそのような $n-1$ 個のフィボナッチ和（フィボナッチ数の和：訳者挿入）の和であるような入力に対してだけである．ゆえに，次のような重大な疑問が生じる：最初の連の数がそのような理想的な和でないときには，どのようにしたらよいであろうか．答は簡単である（そして，このような状況では典形的なものである）．仮想的な空の連を仮定し，本当の連と仮想的な連の和が完全和になるようにすればよい．この空の連は，**空連** (dummy run) と呼ばれる．しかし，これだけでは実際には満足できる答ではない．なぜなら，さらにもっと困難な問題がすぐに生じるからである．すなわちマージの過程で，この空連をどのようにして認識したらよいかということである．この問題に答える前に，連の初期分配の問題について考え，本当の連と空連を $n-1$ 本のテープ上に分配する規則を決定しなければならない．

しかしながら，分配のための適切な規則を見つけるためには，実際の連と空

表 2.15 完全な分配が可能な連の数

l \ n	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001
15	1597	13745	34513	56417	76806	95617
16	2584	25281	66526	110913	152351	190465
17	4181	46499	128233	218049	302201	379399
18	6765	85525	247177	428673	599441	755749
19	10946	157305	476449	842749	1189041	1505425
20	17711	289329	918385	1656801	2358561	2998753

とをマージする方法を知っていなければならない．明らかに，テープ i から空連を選択することは，このマージの過程ではテープ i を無視し，したがって，$n-1$ 本より少ない入力テープからマージすることを意味する．$n-1$ 本の全ての入力テープから空連をマージすることは，実際にはマージ操作は行なわず，その代わり，出力テープに結果としての空連を記録することを意味する．このことから，$n-1$ 本のテープにできるだけ均一に空連を分配すべきであることが結論できる．なぜなら，われわれに興味があるのは，できるだけ多くのテープからの有効なマージだからである．

しばらくの間，空連については忘れ，**未知**の個数の連を $n-1$ 本のテープ上に分配する問題を考えよう．分配が進行していく間に各テープ上の理想的な連の数を表わす次数 $n-2$ のフィボナッチ数を生成できることは明らかである．たとえば $n=6$ と仮定すると，表 2.14 を参照し，インデックス $l=1$ の行（1, 1, 1, 1, 1）の示すように連を分配することから初める．もし，もっと

多くの連が存在すれば，次の行（2，2，2，2，1）に進む．もし，まだ入力が空にならないならば，3番目の行（4，4，4，3，2）に従って分配し，以下同様に行なう．行のインデックスを**レベル**と呼ぶことにする．明らかに連の数が多くなればなる程，フィボナッチ数のレベルも高くなるが，これは偶然にもそれ以後のソートで必要なマージ・パス，あるいは，テープ切換えの回数に等しい．

分配アルゴリズムの最初のものは，したがって，次のように定式化される：

1.　分配の目標を次数 $n-2$，レベル1のフィボナッチ数とする．

2.　設定された目標にしたがって分配する．

3.　もし目標が達成されたら，次のレベルのフィボナッチ数を計算する．そして，その値と直前のレベルの値との差を新しい分配の目標とし，ステップ2に戻る．もし，入力が空になってしまい目標が達成できなかったならば，分配プロセスを停止する．

次のレベルのフィボナッチ数を計算する規則は，その定義（2.44）によって与えられる．したがって，われわれはステップ2に注意を集中することができるが，このステップでは与えられた目標にしたがって，残りの連を $n-1$ 本のテープ上に順次分配しなければならない．また，空連を再び考慮しなければならなくなるのもこのステップである．レベルを上げる場合，差 d_i，$i=1, \cdots, n-1$，によって次の目標を記録すると仮定しよう．ただし，d_i はこのステップでテープ i 上に出力されるべき連の数を表わす．ここで，われわれは次のように考えることができる．まずただちに，d_i 個の空連をテープ i 上に出力し，その後の分配は実際の連による空連の**置換え**とみなす．そして，そのような置換えを行なうごとに，d_i から1を引く．したがって，入力が空になったときには，d_i はテープ i 上の空連の数を表わす．

どのようなアルゴリズムによれば最適な分配が得られるかは知られていないが，次のものは大変良好な方法であることが証明されている．これは，**水平分配**（horizontal distribution, Knuth. vol 3. p. 270参照）と呼ばれるが，この言葉は，$n=6$，レベル4に対する**図 2.16** で示されるように，連をサイロの形に積み上げると考えれば理解できる（表 2.14参照）．

残りの空連をできるだけ早く平等に分配するために，水平レベルの連を左から右の順に取り去って実際の連で置き換えることにより，（連の）積上げの高さを減少させる．このようにして，図 2.16 中の番号で示される順番にしたがって，連はテープ上に分配される．

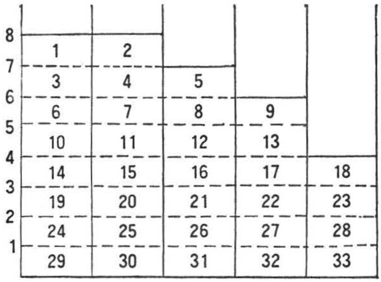


図 2.16 連の"水平分配"

さて，次にこのアルゴリズムを *selecttape* という名前の手続きの形で記述する．この手続きは，1つの連がコピーされ，次の連のための新しいテープが選択される各時点で呼び出される．現在の出力テープのインデックスを表わす変数 j の存在を仮定する．また a_i と d_i は，テープ i に対する理想的および仮の分配数を表わす．

$$\begin{aligned}&j\colon tapeno;\\&a, d\colon \textbf{array}\ [tapeno]\ \textbf{of}\ index;\\&level\colon integer\end{aligned} \qquad (2.45)$$

これらの変数には，次のような初期値が与えられる．

$$\begin{aligned}a_i &= 1, & d_i &= 1 \quad (i = 1, \cdots, n-1\ \text{に対して})\\ a_n &= 0, & d_n &= 0 \quad (\text{仮に})\\ j &= 1\\ level &= 1\end{aligned}$$

selecttape は，レベルが増加するごとに，表 2.14 の次の行，すなわち値 $a_1^{(l)}$, …, $a_{n-1}^{(l)}$ を計算する．また，そのときに，"次の目標" すなわち差 $d_i = a_i^{(l)} - a_i^{(l-1)}$ も計算される．次に示すアルゴリズムは，結果の d_i がインデックスに関して減少するという事実（図 2.16 の下り階段）を利用している．（レベル 0 からレベル 1 に移るときは例外であることに注意せよ．したがって，このアルゴリズムはレベル 1 を出発点として使わなければならない）．*selecttape* は d_j を 1 だけ減らすことによって終わるが，この操作はテープ j 上の空連を実際の連で置き換えることに対応する．

```
procedure selecttape;
   var i: tapeno; z: integer;
begin
   if d[j] < d[j+1] then j := j+1 else
   begin if d[j] = 0 then
      begin level := level + 1; z := a[1];
         for i := 1 to n-1 do
         begin d[i] := z+a[i+1]-a[i]; a[i] := z+a[i+1]          (2.46)
         end
      end ;
      j := 1
   end ;
   d[j] := d[j] -1
end
```

入力テープ $f0$ から $f[j]$ に1つの連をコピーする手続きが利用できるとすれば，初めの分配フェーズは次のように定式化できる（入力テープは常に少なくとも1つの連は含んでいると仮定する）：

```
repeat selecttape; copyrun
until eof (f0)                                                  (2.47)
```

しかし，われわれはここでしばらく休んで，前に論じた自然なマージ・アルゴリズムでの連の分配の際に出合った現象を思い出そう：すなわち，同じ出力テープに連続して到着した2つの連が1つの連になり，その結果，仮定した連の数が正しくなくなることである．この場合には，ソート・アルゴリズムを工夫して，その正当性が連の数に依存しないようにすることにより，この副作用を安全に避けることができた．しかしながら，ポリフェーズ・ソートの場合には，各テープ上の連の数を正確に管理することに特に気をつけているので，そのような偶然のマージの結果を見過ごすことはできない．

したがって，分配アルゴリズムがより複雑になることは避けられない．各テープ上の最後の連の最後の項目のキーを記憶しておくことが必要になるが，このために次の変数を導入する．

```
last: array [tapeno] of integer
```

このとき，分配アルゴリズムは次のように書かれる．

```
repeat selecttape;
    if last[j] ≤ f0↑ .key then
        "continue old run";                              (2.48)
    copyrun; last[j] := f0↑ .key
until eof(f0)
```

このアルゴリズムの明らかな誤りは，最初の連がコピーされるまで *last*[*j*] の値が得られない，すなわち定義されないことを忘れたことである．正しい解は，まず *last*[*j*] の値を調べずに $n-1$ 本の各テープ上に1つの連を分配する．残りの連は (2.49) にしたがって分配される．

```
while ¬eof(f0) do
begin selecttape;
    if last[j] ≤ f0↑ .key then
    begin {continue old run}
        copyrun;                                         (2.49)
        if eof(f0) then d[j] := d[j] + 1 else copyrun
    end
    else copyrun
end
```

ここでは，*last*[*j*] への代入は，手続き *copyrun* の中に含まれていると仮定している．

さて，いよいよ目的のポリフェーズ・ソート・アルゴリズムの構成にとりかかろう．その主要な構造は，n ウエイ・マージ・プログラムの主要部に似ている：外側のループでは，入力テープが空になるまで連をマージし，内側のループでは，各入力テープから1つの連をマージし，そして一番内側のループでは，最初のキーを選び，付随する項目を出力ファイルに転送する．主な相違点は次のとおりである：

1. $n/2$ 本ではなく，ただ1本の出力テープが各パスに存在する．
2. 各パスの後で $n/2$ 本の入力テープと $n/2$ 本の出力テープを切り換える代わりに，テープを回転させる．これは，インデックス写像 t を用いて行なわれる．
3. 入力テープの本数はマージごとに変わる：各マージの初めに，その数は空連のカウント数 d_i から決定される．もし，すべての i に対して $d_i > 0$ ならば，$n-1$ 本の空連が仮りにマージされて1つの空連になるが，それは単に出力テープのカウント数 d_n を増加させるだけである．そうでない

ときには，$d_i=0$ であるすべてのテープから1つの連がマージされ，そして他のすべてのテープから1つの空連が取り去られたことを示すために，それらのテープの d_i は減らされる．あるマージに参加した入力テープの本数を k で表わす．

4. $n-1$ 本目のテープの end-of-file（ファイルの終わり）状態によってフェーズの終了とすることはできない．なぜなら，そのテープからの空連をまじえたマージが，さらに必要であるかも知れないからである．その代わり，理論的に必要なマージの回数は，係数 $a_i^{(l)}$ から決定される．係数 $a_i^{(l)}$ は分配フェーズ中で計算されたが，マージを行なうときには“逆向き”に再計算することが可能である．

ポリフェーズ・ソートの主要な部分は，これらの規則に従って次のように定式化することができる．ただし，初期連の入っている $n-1$ 本のテープはすべてリセットされており，またテープ写像は $t_i=i$ に初期設定されていると仮定する．

```
repeat {merge from t[1] ... t[n-1] to t[n]}
    z := a[n-1]; d[n] := 0; rewrite(f[t[n]]);
    repeat k := 0;  {merge one run}
        {determine no. k of active input tapes}
        for i := 1 to n-1 do
        if d[i] > 0 then d[i] := d[i] -1 else
            begin k := k+1; ta[k] := t[i]
            end ;
        if k = 0 then d[n] := d[n] + 1 else
            "merge one real run from ta[1] ... ta[k]";        (2.50)
        z := z-1
    until z = 0;
    reset(f[t[n]]);
    "rotate tapes in map t; compute a[i] on next level";
    rewrite(f[t[n]]); level := level - 1
until level = 0;
{sorted output is on t[1]}
```

実際のマージ操作は，n ウエイ・マージのプログラムとほとんど同一であり，ただテープ除去のアルゴリズムがいくらか簡単になっているところが異なるだけである．テープ・インデックス写像および対応するカウント数 d_i の回転（そして係数 $a_i^{(l)}$ の下向きの再計算）は簡単であり，その詳細は，ポリフェーズ・

アルゴリズム全体を表現した**プログラム 2.16** から知ることができる.

プログラム 2.16 ポリフェーズ・ソート

```
program polysort (output);
{polyphase sort with n tapes}
const n = 6;                {no. of tapes}
type item = record
              key: integer
            end ;
   tape = file of item;
   tapeno = 1 .. n;
var leng, rand: integer;        {used to generate file}
   eot: boolean;
   buf: item;
   f0: tape;  {f0 is the input tape with random numbers}
   f: array [1 .. n] of tape;
procedure list (var f: tape; n: tapeno);
   var z: integer;
begin z := 0;
   writeln ('TAPE', n: 2);
   while ¬eof(f) do
   begin read(f, buf); write(output, buf.key: 5); z := z+1;
      if z = 25 then
         begin writeln (output); z := 0
         end
   end ;
   if z ≠ 0 then writeln (output); reset(f)
end {list} ;
procedure polyphasesort;
   var i,j,mx,tn: tapeno;
      k, level: integer;
      a, d: array [tapeno] of integer;
         {a[j] = ideal number of runs on tape j}
         {d[j] = number of dummy runs on tape j}
      dn,x,min,z: integer;
      last: array [tapeno] of integer;
         {last[j] = key of tail item on tape j}
      t,ta: array [tapeno] of tapeno;
         {mappings of tape numbers}
```

```
procedure selecttape;
    var i: tapeno; z: integer;
begin
    if d[j] < d[j+1] then j := j+1 else
    begin if d[j] = 0 then
            begin level := level + 1; z := a[1];
                for i := 1 to n-1 do
                begin d[i] := z + a[i+1] - a[i]; a[i] := z + a[i+1]
                end
            end ;
            j := 1
    end ;
    d[j] := d[j] -1
end ;
procedure copyrun;
begin {copy one run from f0 to tape j}
    repeat read(f0, buf); write(f[j], buf);
    until eof(f0) ∨ (buf .key > f0↑ .key);
    last[j] := buf .key
end ;
begin {distribute initial runs}
    for i := 1 to n-1 do
        begin a[i] := 1; d[i] := 1: rewrite(f[i])
        end ;
    level := 1; j := 1; a[n] := 0; d[n] := 0;
    repeat selecttape; copyrun
    until eof(f0) ∨ (j=n-1);
    while ¬eof(f0) do
    begin selecttape;
        if last[j] ≤ f0↑ .key then
        begin {continue old run}
            copyrun;
            if eof(f0) then d[j] := d[j] + 1 else copyrun
        end
        else copyrun
    end ;
```

```
for i := 1 to n-1 do reset(f[i]);
for i := 1 to n do t[i] := i;
repeat {merge from t[1] ... t[n-1] to t[n]}
  z := a[n-1]; d[n] := 0; rewrite(f[t[n]]);
  repeat k := 0;      {merge one run}
    for i := 1 to n-1 do
    if d[i] > 0 then d[i] := d[i]-1 else
      begin k := k+1; ta[k] := t[i]
      end ;
    if k = 0 then d[n] := d[n] + 1 else
    begin {merge one real run from ta[1] ... ta[k]
      repeat i := 1; mx := 1;
        min := f[ta[1]]↑ .key;
        while i < k do
        begin i := i+1; x := f[ta[i]]↑ .key;
          if x < min then
          begin min := x; mx := i
          end
        end ;
        {ta[mx] contains minimal element; move it to t[n]}
        read(f[ta[mx]], buf); eot := eof(f[ta[mx]]);
        write(f[t[n]], buf);
        if (buf.key > f[ta[mx]]↑.key) ∨ eot then
        begin {drop this tape}
          ta[mx] := ta[k]; k := k-1
        end
      until k = 0
    end ;
    z := z-1
  until z = 0;
  reset(f[t[n]]); list(f[t[n]], t[n]); {rotate tapes}
  tn := t[n]; dn := d[n]; z := a[n-1];
  for i := n downto 2 do
    begin t[i] := t[i-1]; d[i] := d[i-1]; a[i] := a[i-1] - z
    end ;
  t[1] := tn; d[1] := dn; a[1] := z;
  {sorted output is on t[1]}
  list(f[t[1]], t[1]); level := level - 1
 until level = 0;
end {polyphasesort} ;
```

```
begin {generate random file}
   leng := 200; rand := 7789;
   repeat rand := (131071*rand) mod 2147483647;
      buf .key := rand div 2147484; write(f0, buf); leng := leng - 1
   until leng = 0;
   reset(f0); list(f0, 1);
   polyphasesort
end
```

2.3.5　初期連の分配

順次的ソーティング・プログラムの構造は複雑なものになってしまった．これに反し，配列を操作する方法はもっと簡単であるが，これはソートすべき全データを保持するだけの十分に大きなランダム・アクセス記憶装置の利用を前提としていたからである．ほとんどの場合そのような記憶装置は利用不可能であり，その代わりにテープのような十分に大きな順次的記憶装置を使用しなければならない．これまで構成してきた順次的ソーティング法は，ファイル・バッファと，当然ではあるがプログラム自身の他には何ら主記憶を必要としないことに注意してほしい．しかしながら，小さな計算機でもほとんどの場合，ここで構成したプログラムに必要なものより多量のランダム・アクセス主記憶容量をもっており，これを最適に利用する方法を考えるべきである．

それに対する解は，配列とファイルのソーティング技法を結合することにある．特に，配列のソートを初期連の分配フェーズに適用することができるが，これにより，これらの連は利用可能な主記憶の大きさに近似的に等しい長さ l を初めから持つことができる．それ以後のマージ・パスでは，連の長さは確実に増加しつづけ利用可能な主記憶より大きくなるので，これ以上配列ソートを行なっても性能を改善できないことは明らかである．この結果，幸運にもわれわれは初期連の生成アルゴリズムの改良に注意を集中することができる．

当然，対数的な配列ソーティング法のみを考慮すればよいことは明らかである．それらのうちで最も適したものは，木ソートまたはヒープ・ソートである(2.2.5節参照)．ヒープは，すべてのファイル要素が，早いか遅いかの差はあるにしても，通過しなければならないトンネルと見ることができる．最小のキーはヒープの頂上からただちに取り除かれ，その置換えは非常に能率的に行なわれる．1つの要素を“ヒープ・トンネル” h を通して，入力テープ $f0$ から出力テープ $f[j]$ に転送する操作は，次のように簡単に記述できる．

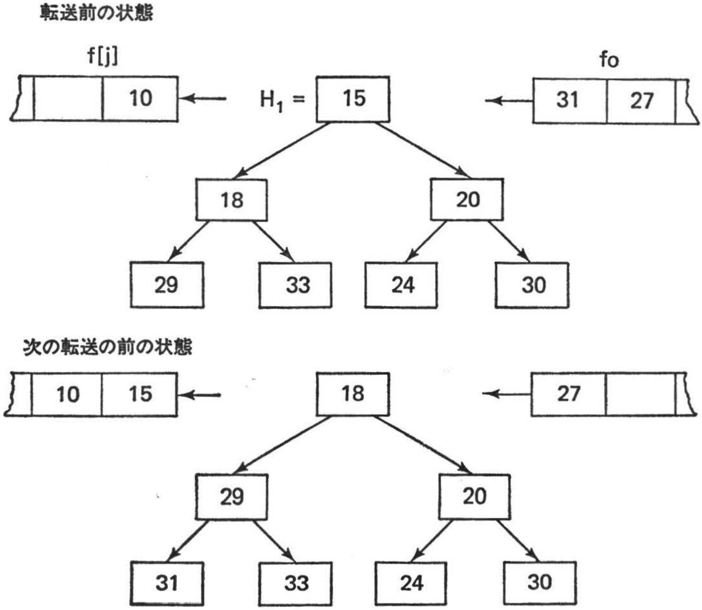


図 2.17 ヒープ中のキーのシフト

$$
\begin{aligned}
&write(f[j], h[1]);\\
&read(f0,\ h[1]);\\
&shift(1, n)
\end{aligned}
\tag{2.51}
$$

"*shift*" は，新しく挿入された成分 $h[1]$ を適当な場所にシフト・ダウンするための手続きであり，2.2.5 節で述べられている．$h[1]$ はそのヒープ上の**最小**の項目であることに注意せよ．**図 2.17** に 1 つの例を示す．

このプログラムは最終的には次のような理由により相当複雑になる．

1. ヒープ h は初めは空であるので，まずいっぱいに満たさなければならない．
2. 終わりの頃には，ヒープはただ部分的に満たされているだけであり，そして最終的には空になる．
3. 出力テープのインデックス j を正しい時点で変更するために，新しい連の始めを憶えておかなければならない．

先に進む前に，明らかに必要となる変数を形式的に宣言しておく．

$$
\begin{array}{l}
\textbf{var}\ f0\text{: } tape;\\
\quad f\ \text{: }\textbf{array}\ [tapeno]\ \textbf{of}\ tape;\\
\quad h\ \text{: }\textbf{array}\ [1\,..\,m]\ \textbf{of}\ item;\\
\quad l,r\text{: } integer
\end{array}
\qquad (2.52)
$$

ここで，m はヒープ h の大きさである．$m/2$ を示すために定数 mh を用いる．また，l と r は h 上のインデックスである．通過のプロセスは5つの異なった部分に分けることができる．

1. $f0$ から初めの mh 個の項目を読み，キーの間の順序が規定されていないヒープの上半分[†]に置く．
2. 残りの mh 個の項目を読み，ヒープの下半分に置き，それぞれ適当な位置にシフトさせる（ヒープの構成）．
3. l の値を m として，$f0$ 上の残りの項目すべてに対して次のステップを繰り返す：$h[1]$ を適当な出力テープに出力する．もし，そのキーが入力テープ上の次の項目のキーより小さいか又は等しいならば，この次の項目は同じ連に属し，したがって適当な位置にシフトされ得る．もしそうでないなら，ヒープの大きさを減らし，次の連を入れるために成長させた2番目の，“上側”のヒープにこの新しい項目を置く．2つのヒープの境界をインデックス l で示す．したがって，“下側”の，あるいは現在のヒープは項目 $h[1]\cdots h[l]$ から成り，“上側”の，あるいは次のヒープは $h[l+1]\cdots h[m]$ から成る．もし $l=0$ ならば，出力テープを変え，l を m にリセットする．
4. 入力テープが空になったとする．まず，r を m に等しくし，現在の連を終端させるために下側の部分を出力する．それと同時に上側の部分を成長させそれを徐々に位置 $h[l+1]\cdots h[r]$ に移し換える．
5. 最後の連をヒープ中の残りの項目から生成する．

以後，この5つの部分を完全なプログラムとして詳述するが，このプログラムは，連の終わりが検出され出力テープのインデックスを変更する動作を起動する際には常に手続き *selecttape* を呼び出す．しかし，**プログラム 2.17** ではその代わりに，見せかけだけの手続きが用いられている．この手続きは単に生成された連の数を数えるだけであり，すべての要素はテープ $f1$ に出される．

このプログラムを，たとえばポリフェーズ・ソートと結合しようとすると，

---

† **訳注**　ヒープを配列で表現したときインデックスに関して上半分すなわち右半分

プログラム 2.17 ヒープによる初期連の分配

```
program distribute(f0,f1,output);
{initial distribution of runs by heap sort}
const m = 30; mh = 15; {size of heap}
type item = record
              key: integer
            end ;
   tape = file of item;
   index = 0 .. m;
var l,r: index;
   f0,f1: tape;
   count: integer;      {counter of runs}
   h: array [1 .. m] of item; {heap}

procedure selecttape;
begin count := count + 1;
   {dummy; count number of distributed runs}
end {selecttape} ;

procedure sift(l,r: index);
   label 13;
   var i,j: integer: x: item;
begin i := l; j := 2*i; x := h[i];
   while j ≤ r do
   begin if j < r then
            if h[j] .key > h[j+1] .key then j := j+1;
         if x .key ≤ h[j] .key then goto 13;
         h[i] := h[j]; i := j; j := 2*i
   end ;
13: h[i] := x
end ;
begin {create initial runs by heapsort}
   count := 0; reset(f0); rewrite(f1);
   selecttape;
  {step 1: fill upper half of heap h}
   l := m;
   repeat read(f0, h[l]); l := l-1
   until l = mh;
  {step 2: fill lower half of heap h}
   repeat read(f0, h[l]); sift(l,m); l := l-1
   until l = 0;
```

```
  {step 3: pass runs through full heap}
   l := m;
   while ¬eof(f0) do
   begin write(f1, h[1]);
       if h[1] .key ≤ f0↑ .key then
       begin {new record belongs to same run}
           read(f0, h[1]); sift(1,l);
       end else
       begin {new record belongs to next run}
           h[1] := h[l]; sift(1,l−1);
           read(f0, h[l]); if l ≤ mh then sift(l,m); l := l−1;
           if l = 0 then
           begin {heap is full; start new run}
               l := m; selecttape;
           end
       end
   end ;
  {step 4: flush lower part of heap}
   r := m;
   repeat write(f1, h[1]);
       h[1] := h[l]; sift(1,l−1);
       h[l] := h[r]; r := r−1;
       if l ≤ mh then sift(l,r); l := l−1
   until l = 0;
  {step 5: flush upper part of heap. generate last run}
   selecttape;
   while r > 0 do
   begin write(f1,h[1]);
       h[1] := h[r]; sift(1,r); r := r−1
   end ;
   writeln (count)
end
```

重大な困難に出合う．これは次のような状況から生じる：そのソート・プログラムは，その初めの部分にテープ切換えを行なうためのかなり複雑なルーチンを含んでおり，選ばれたテープにちょうど1つの連を転送する手続き *copyrun* を使用している．一方，このヒープ・ソート・プログラムも複雑なルーチンであり，新しいテープを選択するためだけの閉じた手続き *selecttape* を使用して

いる.

もし，これらの一方（あるいは両方）のプログラムにおいて，その必要な手続きが1個所でしか呼び出されなければ何ら問題はない．しかし，それらの手続きは両方のプログラム中でともに，複数個の場所で呼び出されている．

この状況は，いわゆる**コルーチン**（coroutine）を使うことによって最も良く表わされる；コルーチンは，複数個のプロセスが共存する場合に適している．その最も典形的な場合は，情報の流れを生産するプロセスと，この流れを消費（処理）するプロセスの結合である．この生産者 - 消費者の関係は，2つのコルーチンによって表現することができる．それらのうちどちらも，それ自身で主プログラムになることができる．

コルーチンは，1つ以上の中断点を含んだ手続きまたはサブルーチンと考えることができる．そのような中断点に出合った時には，制御はこのコルーチンを呼び出したプログラムにもどる．そして，このコルーチンが再び呼ばれたときには，実行はその中断点から再開される．われわれの例では，ポリフェーズ・ソートを主プログラムと考えてよく，コルーチンの形に作られた *copyrun* を呼び出す．それは，プログラム2.17の主要部分からなるが，*selecttape* の各呼出しは，この場合中断点を表わす．ファイルの終わりの判定は，コルーチンがその終わりに達したかどうかの判定によって組織的に置き換えられなければならないであろう．論理的に健全な方法は，*eof*(*f*0)の代わりに *eoc*(*copyrum*)とすることである．

解析と結論　連の初期分配をヒープ・ソートによって行なうポリフェーズ・ソートからどのような性能が期待されるであろうか．ヒープの導入によって期待される改良から論じよう．

キーがランダムに分布している列では，連の長さの期待値は2である．この列を大きさ m のヒープを通したら，この長さはいくつになるであろうか．m であると答えやすいかも知れないが，幸運にも確率論的解析による実際の結果は $2m$ である（Knuth, vol. 3, p. 254 参照）．したがって，改良度の期待値は m である．

ポリフェーズ法の性能の推定値は**表2.15**から知ることができるが，この表には，与えられた本数のテープを使って，与えられた数の部分パス（レベル）でソートすることのできる初期連の最大値が示されている．1例として，$n=6$ テープ，ヒープの大きさ $m=100$ のとき，165,860,100個の初期連からなるファイルは，20部分パスでソートすることができる．これは著しい性能であ

る．

ポリフェーズ・ソートとヒープ・ソートの結合について見直してみると，そのプログラムの複雑さに驚かざるを得ない．それにもかかわらず，それの行なう内容は，単純な配列ソーティングの原理に基づいた簡単なプログラムと同じであり，項目の再順序づけという容易に定義される仕事である．本章全体の教訓は，次のことを示すことであると考えてもよい．

1. アルゴリズムとその基礎となるデータ構造との間の緊密な関係，特に後者の前者に対する影響．
2. データに対して利用可能な構造が，（配列の代りに列である場合のように）その仕事にむしろ向いていない場合に対してでも，工夫によってはプログラムの性能が改善できるということ．

演習問題

2.1. プログラム 2.1. から 2.6, 2.8, 2.10 および 2.13 によって与えられるアルゴリズムのうちで，安定なソーティング法はどれか．

2.2. プログラム 2.2 は，while 節中の $l \leq r$ を $l < r$ で置き換えても正しく動作するか．また，文 $r_i = m-1$ と $l_i = m+1$ を $r_i = m$ と $l_i = m$ と簡単化してもまだ正しいか．もしそうでないならば，変更されたプログラムが正しく動作しないような値の集合 $a_1, \cdots, a_n$ を求めよ．

2.3. 諸君の計算機上で3つの単純ソーティング法のプログラムを作り，実行時間を測定せよ．そして実際の時間の推定値を与えるように，C, M に乗ずべき係数を決定せよ．

2.4. 種々のランダムな入力列について，ヒープ・ソート・プログラム 2.8 をテストし，文 **goto** 13 が実行される平均回数を決めよ．この数は比較的小さいので，次の質問は興味がある：while ループから次のテストを取り出す方法があるか．

$$x.key \geq a[j].key$$

2.5. 分割プログラム 2.9 に関連して，次のような“明らかな”プログラムについて考える．

```
i := 1; j := n;
x := a[(n+1) div 2].key;
```

```
repeat
    while a[i].key < x do i := i+1;
    while x < a[j].key do j := j-1;
    w := a[i]; a[i] := a[j]; a[j] := w
until i > j
```

このプログラムが正しく動作しないような値の集合 $a_1, \cdots, a_n$ を見つけよ.

2.6. クイック・ソートとバブル・ソートを次のように結合したプログラムを書け：長さ $m(1 \leq m \leq n)$ の（ソートされていない）分割部分を得るためにクイック・ソートを用い，次にバブル・ソートを用いてソートを完了する．後者では，"管理作業"の量を最小にするために，n 個の要素からなる配列全体を走査するであろうことに注意せよ．全ソート時間を最小にするような m の値を見つけよ．

注意：明らかに，m の最適値は非常に小さい．したがって，それ以上の交換が不必要であることを調べるための最後のパスを含めずに配列を正確に $m-1$ 回走査しても引き合う．

2.7. バブル・ソートの代わりに単純選択ソートを用いて，演習問題2.6と同じことを実行せよ．当然，選択ソートでは配列全体を走査することはできない．したがって，インデックス操作回数の期待値はいくらか多くなる．

2.8. 長さが短い方の分割部分のソートを長い方のソートより先に実行するという方針にしたがって再帰的クイック・ソートのアルゴリズムを書け．前者の作業は繰返し文によって，また後者は再帰呼出しによって行なえ．（したがってプログラム2.10では2個，プログラム2.11では0個の再帰呼出しを含むのに対し，諸君のプログラムには1個の再帰呼出しが含まれることになる）．

2.9. クイック・ソートが最悪（最良）のふるまいをするようなキー $1, 2, \cdots, n$ の置換を見つけよ（$n = 5, 6, 8$）．

2.10. 単純マージ・プログラム2.13と同様に，2倍の長さの配列上をその両端から内側に向かって動作するような自然なマージ・プログラムを構成せよ：その性能をプログラム2.13と比較せよ．

2.11. （2ウェイの）自然なマージでは，対象となるキーの中から最小な値を常に選んでいるわけではなく，一方の連の終わりに達したならば他方の連の残りは出力列に単にコピーされるだけである．たとえば，

2,　4,　5,　1,　2，…
3,　6,　8,　9,　7，…

をマージすると，

2,　3,　4,　5,　6,　8,　9,　1,　2，…

のようになり，

2,　3,　4,　5,　1,　2,　6,　8,　9，…

とはならないが，この方がより良く順序づけられているように見える．この理由は何か．

2.12. プログラム 2.15 で変数 *ta* の役割はなにか．どのような状況のもとで文

begin *rewrite*(*f* [*ta*[*mx*]]) ; …

が実行され，また文

begin *tx* := *ta*[*m*x] ; …

はいつ実行されるか．

2.13. ポリフェーズ・ソート・プログラム 2.16 では変数 *last* が必要であるのに，プログラム 2.15 ではそうでない理由は何か．

2.14. ポリフェーズ法に類似なソーティング法に**カスケード・マージ**(cascade merge) 法がある [2.1 および 2.9]．この方法は異なるマージ・パターンを利用する．たとえば，6 本のテープ $T1$，…，$T6$ が与えられたとき，カスケード・マージでも $T1$，…，$T5$ に連を“完全に分配”することから始め，$T5$ が空になるまで $T1$，…，$T5$ から $T6$ 上に 5 ウェイ・マージを行なう．次に，($T6$ を使用せずに) $T5$ 上に 4 ウェイ・マージし，さらに $T4$ 上に 3 ウエイ・マージ，$T3$ 上に 2 ウェイ・マージを行ない，最後に $T1$ から $T2$ 上にコピー操作を行なう．次のパスでは同様に $T1$ 上への 5 ウエイ・マージから始め，以下これを繰り返す．この方式は，テープが使われずに遊んでいる時間があること，および単純なコピー操作が含まれるので，ポリフェーズ法より劣っているように見える．しかし驚いたことには，(非常に) 大きなファイルや 6 本以上のテープに対してはポリフェーズ法より優れている．カスケード・マージの原理による良く構造化されたプログラムを書け．

参考文献

2-1. Betz, B. K. and Carter, *ACM National Conf.*, **14**, (1959), Paper 14.

2-2. Floyd, R. W., "Treesort" (Algorithms 113 and 243), *Comm. ACM*, **5**, No. 8 (1962), 434, and *Comm. ACM*, **7**, No. 12 (1964), 701.

2-3. Gilstad, R. L., "Polyphase Merge Sorting—An Advanced Technique," *Proc. AFIPS Eastern Jt. Comp. Conf.*, **18**, (1960), 143–48.

2-4. Hoare, C. A. R., "Proof of a Program: FIND," *Comm. ACM*, **13**, No. 1 (1970), 39–45.

2-5. _____, "Proof of a Recursive Program: Quicksort," *Comp. J.*, **14**, No. 4 (1971), 391–95.

2-6. __________, "Quicksort," *Comp. J.*, **5**, No. 1 (1962), 10–15.

2-7. Knuth, D. E., *The Art of Computer Programming*, Vol. 3 (Reading, Mass.: Addison-Wesley, 1973).

2-8. _____, *The Art of Computer Programming*, **3**, pp. 86–95.

2-9. _____, *The Art of Computer Programming*, **3**, p. 289.

2-10. Lorin, H., "A Guided Bibliography to Sorting," *IBM Syst. J.*, **10**, No. 3 (1971), 244-54.

2-11. Shell, D. L., "A Highspeed Sorting Procedure," *Comm. ACM*, **2**, No. 7 (1959), 30–32.

2-12. Singleton, R. C., "An Efficient Algorithm for Sorting with Minimal Storage" (Algorithm 347), *Comm. ACM*, **12**, No. 3 (1969), 185.

2-13. Van Emden, M. H., "Increasing the Efficiency of Quicksort" (Algorithm 402), *Comm. ACM*, **13**, No. 9 (1970), 563–66, 693.

2-14. Williams, J. W. J., "Heapsort" (Algorithm 232), *Comm. ACM*, **7**, No. 6 (1964), 347–48.

第3章

再帰的アルゴリズム

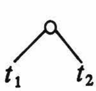
3.1 はじめに

ある物は，それが自分自身を部分的に含んでいたり，それを用いて定義されているとき，**再帰的**(recursive)であるといわれる．**再帰** (recursion) は数学だけでなく日常生活でも出合うことがある．誰でも，それ自身を含んだような広告の絵を見たことがあるであろう (**図 3.1**)．

再帰は，数学的定義においては特に強力な方法である．よく知られた例としては，自然数，木構造やある種の関数などの定義がある．

1. 自然数：
 (a) 1は自然数である．
 (b) ある自然数の直後の数も自然数である．

2. 木構造：
 (a) ○は木である（空木と呼ばれる）．
 (b) もし，t_1 と t_2 が木であるならば，



 も木である（上下逆さまに描いてある）．

3. 階乗関数 $n!$（非負整数に対して）：
 (a) $0! = 1$
 (b) $n > 0$ ならば，$n! = n\cdot(n-1)!$

明らかに，再帰の能力は有限の記述によって無限個の物を定義できるところにある．同様に，無限に多くの計算は，たとえ明確な繰り返しがそのプログラム

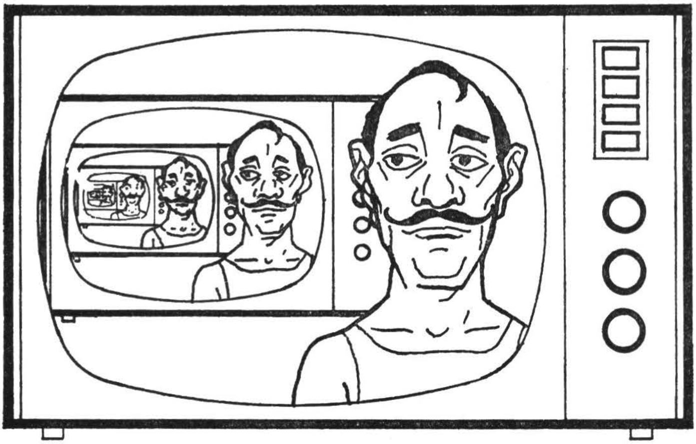


図 3.1 再帰的な絵

に含まれていなくても，有限な再帰的プログラムによって記述することができる．しかしながら，再帰的アルゴリズムが特に適しているのは，解くべき問題や計算すべき関数，あるいは処理すべきデータ構造がすでに再帰的に定義されている場合である．一般に，再帰的プログラム P は，(P を含まない) 基本となる文 S_i と P 自身の合成 $\mathcal{P}$ として表わすことができる．

$$P \equiv \mathcal{P}[S_i, P] \tag{3.1}$$

プログラムを再帰的に表現するための必要にしてかつ十分な道具は，**手続き**や**サブルーチン**である．なぜなら，それは文に名前をつけることができ，これによってその文を起動することができるからである．手続き P でそれ自身が陽に参照されているとき，それは**直接的に再帰的**といわれる．一方，もし P が他の手続き Q の参照を含み，Q が (直接的にあるいは間接的に) P を参照しているとき，P は**間接的に再帰的**といわれる．したがって，再帰が使用されているかどうかは，そのプログラム・テキストからただちに明らかというわけではない．手続きには，それに局所的なものの集合を対応させるのが普通である．それらは変数，定数，型，あるいは手続きの集合であり，この手続きに対して局所的に定義され，この手続きの外では存在せず意味をもたないものである．そのような手続きが再帰的に呼び出されるごとに，限定された局所的変数の新しい集合が生成される．もちろん，それらは，手続きのそれ以前の出現に対して局所的な集合中の対応する要素と同じ意味をもつものではあるが，それらの

値は異なっており，名前の混同は識別名の有効範囲に関する規則によって回避される．すなわち識別名は，常に最も最近に生成された変数の集合を指す．この規則はその定義から，その手続きに付随した手続きパラメータに対しても適用される．

繰返し文と同様に，再帰的手続きによって停止しない計算の可能性が生じ，したがって，**停止**問題を考える必要が生じる．明らかに，基本的に要求されることは，満たされないことがあるようなある条件 B のもとで手続き P の再帰的呼出しを行なうことである．それゆえ再帰的アルゴリズムの形はもっと正確には，

$$P \equiv \textbf{if } B \textbf{ then } \mathcal{P}[S_i, P] \tag{3.2}$$

または

$$P \equiv \mathcal{P}[S_i, \textbf{if } B \textbf{ then } P] \tag{3.3}$$

のように表わすことができる．

繰返しが停止することを示す基本的な手法は，$f(x) \leq 0$ が（while や repeat 節の）停止を意味するような関数 $f(x)$（x はそのプログラム中の変数の集合とする）を定義し，繰返しが実行されるごとに $f(x)$ が減少することを証明することである．同様にして，P の実行ごとに $f(x)$ が減少することを示すことによって，再帰的プログラムの停止を証明することができる．停止を保証するための特に明らかな方法は，P に(値)パラメータ，たとえば n，を関連させ，$n-1$ をパラメータ値として P を再帰的に呼び出すことである．これは次のプログラム図式によって表現することができる．

$$P(n) \equiv \textbf{if } n > 0 \textbf{ then } \mathcal{P}[S_i, P(n-1)] \tag{3.4}$$

$$P(n) \equiv \mathcal{P}[S_i, \textbf{if } n > 0 \textbf{ then } P(n-1)] \tag{3.5}$$

実際の応用では，再帰の究極的な深さが有限であるばかりでなく，実際には小さいことを示すことが特に重要である．その理由は，手続き P の各再帰的起動に際して，その変数を収容するためにある量の記憶領域が必要となるからである．これらの局所的な束縛変数に加えて，計算に関する現在の状態を記憶しておき，P の新しい実行が終了し古い実行に戻るときに，それを取り出すことができるようになっていなければならない．われわれはすでに第2章において，手続き *quicksort* を作る場合にこのような状況に出合っている．n 個の項目を2つの部分に分割する文と，この2つの部分をソートするための2つの再帰呼出しから“素直に”プログラムを構成すると，再帰の深さは最悪な場合には n

に近づくことを発見した．一方，状況をより詳細に調査することにより，この深さを $\log n$ にまで近づけることが可能であった．n と $\log n$ の差は，再帰が非常に不適当であるような場合を，それが完全に実用的であるような場合に転換させるに十分なものである．

3.2 再帰を用いるべきでない場合

再帰は，もとの問題や扱うべきデータが再帰的に定義されている場合には特に適している．しかしながら，このことは，再帰的アルゴリズムがその問題を解くための最良の方法であることを，そのような再帰的定義が保証していることを意味するものではない．実際，再帰的アルゴリズムの概念の説明にそのような不適当な例題を用いたことが，プログラミングにおける再帰の利用に対する憂慮と反感を蔓延させ，再帰を非能力と同一視する主な原因であった．この外にも，広く普及しているプログラム言語 FORTRAN ではサブルーチンの再帰的使用を禁止しており，再帰的な解が適当な場合であっても，そのような解が作れないということもあった．

再帰の使用を避けるべきであるようなプログラムは，その構成のパターンを表わすある形式によって特徴づけられる．その形式は，(3.6) あるいは等価ではあるが(3.7)のようなものである．

$$P \equiv \textbf{if } B \textbf{ then} (S \, ; P) \tag{3.6}$$

$$P \equiv (S \, ; \textbf{if } B \textbf{ then } P) \tag{3.7}$$

これらの形式は，簡単な再帰関係によって定義される値を計算する場合には自然なものである．有名な階乗数 $f_i = i!$ の例を調べてみよう．

$$\begin{aligned} i &= 0, 1, 2, 3, \ 4, \ \ 5, \ \cdots \\ f_i &= 1, 1, 2, 6, 24, 120, \ \cdots \end{aligned} \tag{3.8}$$

“0番目” の数は陽に $f_0 = 1$ と定義されているが，それ以後の数は，通常一再帰的一にその直前の数を使って定義される．

$$f_{i+1} = (i+1) \cdot f_i \tag{3.9}$$

この公式は，n 番目の階乗数へ進むための再帰的なアルゴリズムを示唆している．もし値 i と i 番目の再帰のレベルにおける f_i を表わすために 2 つの変数 I と F を導入すれば，数列 (3.8) において次の数に進むために必要な計算は

$$I := I+1; \quad F := I*F \tag{3.10}$$

となることがわかる．そして，(3.6) の S に(3.10)を代入することにより，次の再帰的プログラムが得られる．

$$P \equiv \textbf{if } I < n \textbf{ then } (I := I+1;\ F := I*F\ ;\ P)$$
$$I := 0;\ F := 1;\ P \qquad (3.11)$$

(3.11)の最初の行は，われわれの通常のプログラミング上の記法では次のように表わされる．

```
procedure P;
begin if I < n then
      begin I := I + 1; F := I*F; P
      end
end
```
(3.12)

本質的には等価であるが，より頻繁に使われる形は (3.13) で与えられるものである．P はいわゆる関数手続によって置き換えられているが，これは結果の値が陽に関連づけられている手続きであり，それゆえ，式の構成要素として直接に利用可能なものである．したがって，変数 F は余分である．また，I の役割は手続きの陽なパラメータに与えられている．

```
function F(I: integer): integer;
begin if I > 0 then F := I*F(I - 1)
               else F := 1
end
```
(3.13)

この場合には，再帰は単純な繰返しによって，すなわち次のプログラムによって置き換えることができることは全く明白である．

```
I := 0;  F := 1;
while I < n do
   begin I := I + 1; F := I*F
   end
```
(3.14)

一般に，一般形(3.6)あるいは(3.7)に対応するプログラムは，式(3.15)に対応するものに変換されるべきである．

$$P \equiv (x := x_0;\ \textbf{while } B \textbf{ do } S) \qquad (3.15)$$

もっと複雑な再帰的構成形式をしているが繰返し形式に翻訳することができ，またそのようにすべきであるものが他にもある．たとえば，再帰関係

$$\mathrm{fib}_{n+1} = \mathrm{fib}_n + \mathrm{fib}_{n-1} \quad ただし\ n > 0$$

および $\mathrm{fib}_1 = 1$，$\mathrm{fib}_0 = 0$ によって定義されるフィボナッチ数の計算を考える．直接的で素直な方法によって次のプログラムが得られる．

```
function Fib(n: integer): integer;
begin if n = 0 then Fib := 0 else
      if n = 1 then Fib := 1 else
      Fib := Fib(n-1) + Fib(n-2)
end                                          (3.17)
```

関数呼出し $Fib(n)$ による fib_n の計算は，この関数手続きを再帰的に起動させる．その回数は何回であろうか．$n > 1$ の場合の各呼出しは，さらに2回の呼出しを起こす．すなわち，呼出しの全回数は指数関数的に増加する（**図 3.2** 参照）．そのようなプログラムは，明らかに非実用的である．

しかし，補助変数 $x = \mathrm{fib}_i$，$y = \mathrm{fib}_{i-1}$ を使うことにより，同じ値の再計算を行なわないような繰返し形式によって，フィボナッチ数が計算可能であることは明白である．

```
{compute x = fib_n for n > 0}
i := 1; x := 1; y := 0;
while i < n do
   begin z := x; i := i + 1;
         x := x + y; y := z
   end                                       (3.18)
```

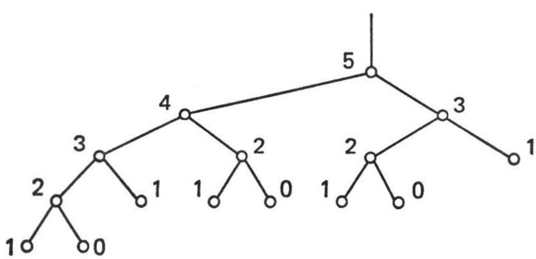


図 3.2　n=5 に対する $Fib(n)$ の 15 回の呼出し

(注意：x, y, z に対する3つの代入は，補助変数 z を用いずに，ただ2つの代入 $x:=x+y$, $y:=x-y$ によって表現することが可能である).

以上の例から得られる教訓は，繰返しによる**明白な**解があるときには，再帰の使用を避けるべきであるということである.

しかしながら，このことはいかなる代価を払っても再帰を使用しないようにすべきである，というものではない．以後の節や章が示すように，再帰の良い応用が数多くある．本質的に非再帰的な計算機上で再帰的な手続きが実現されているという事実は，実際的な目的からは，どんな再帰的プログラムも純粋に繰返し形のプログラムに変換できることを示している．しかし，このためには再帰スタックを陽に操作せねばならず，多くの場合，問題の本質をぼやかし，理解を非常に困難にする．したがって，その本質が繰返しではなく再帰であるようなアルゴリズムは，再帰的手続きとして定式化されるべきであることを学ばねばならない．この点を認識するために，読者はプログラム2.10と2.11を参照し比較してほしい.

この章の残りの部分では，再帰の使用が正しいと認識されるような状況において，いくつかの再帰的プログラムの構成を行なう.

また第4章，第5章でも再帰を非常によく利用するが，これは基本となるデータ構造が再帰的な解の選択を明白で自然なものと思わせるからである.

3.3　再帰的プログラムの2つの例

図3.5に示される魅力的なグラフィック・パターンは，5つの曲線の重ね合わせから構成されている．これらの曲線は規則正しいパターンに従っているので，計算機の制御のもとにプロッタによって描くことができると考えられる．われわれの目標は，プロット・プログラムが構成可能であるような再帰の形を発見することである．調べてみると，重ね合わされた曲線のうちの3つのものは，**図3.3**に示される形をしていることがわかる．それらを H_1, H_2 および H_3 と表わす.

これらの図形から次のことがわかる．すなわち，H_{i+1} は，大きさが半分の4つの H_i を適当に回転させ，3本の結合線でそれらを結ぶことによって構成されている．H_1 も空の H_0 4個を3本の直線で結合したものと考えられることに注意しよう．H_i は，その発明者 D. Hilbert (1891) の名をとって，位数 i の Hilbert 曲線と呼ばれる.

曲線をプロットするための基本的道具として，2つの座標変数 x と y，手続き *setplot* (座標 x と y にペンをセットする) および，手続き *plot* (ペンを現在

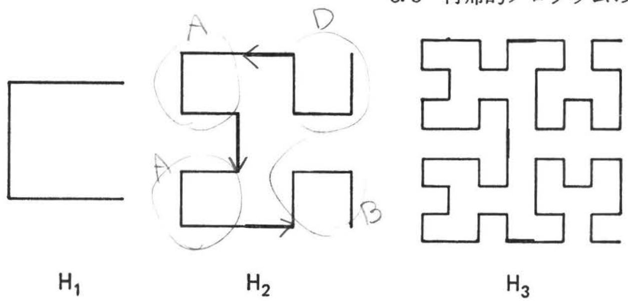


図 3.3　位数 1, 2, 3 の Hilbert 曲線

の位置から x と y で示される位置まで動かす）を仮定しよう．

各曲線 H_i は大きさが半分の4個の H_{i-1} から構成されている．したがって，H_i を描くための手続きが，適当な大きさと傾きの H_{i-1} を4つ描くための部分の合成として表現されることは自然である．この4つの部分を A, B, C, D で表わし，結合線を描くためのルーチンを，対応する方向を指している矢印で表わすことにすれば，次のような再帰図式が得られる（図 3.3参照）．

$$
\begin{aligned}
&A: \quad D \leftarrow A \downarrow A \rightarrow B \\
&B: \quad C \uparrow B \rightarrow B \downarrow A \\
&C: \quad B \rightarrow C \uparrow C \leftarrow D \\
&D: \quad A \downarrow D \leftarrow D \uparrow C
\end{aligned}
\tag{3.19}
$$

もし，単位直線の長さを h で表わすと，図式 A に対応する手続きは，同様に設計される手続き B と D および自分自身の再帰的呼出しを使って容易に表現される．

```
procedure A(i: integer);
begin if i > 0 then
      begin D(i-1); x := x-h; plot;
            A(i-1); y := y-h; plot;
            A(i-1); x := x+h; plot;
            B(i-1)
      end
end
```
(3.20)

この手続きは，重ね合わすべきどの Hilbert 曲線に対しても1回だけ主プログラムによって起動される．主プログラムは，曲線の出発点，すなわち x と y の初期値，および単位増分 h を決定する．$h0$ はページの全長を表わし，適

当な $k \geq n$ に対して，$h0 = 2^k$ を満たさなければならない (**図 3.4** 参照)．全体のプログラムは，n 本の Hilbert 曲線 H_1, …, H_n を描く (**プログラム 3.1** および図 3.5 参照)．

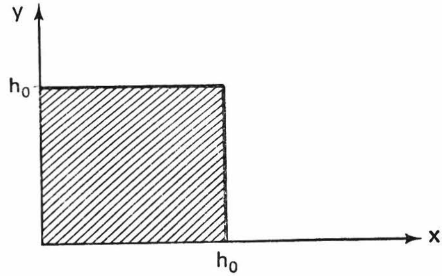


図 3.4　単位枠

プログラム 3.1　Hilbert 曲線

```
program Hilbert(pf,output);
{plot Hilbert curves of orders 1 to n}
const n = 4; h0 = 512;
var i,h,x,y,x0,y0: integer;
   pf: file of integer;      {plot file}
procedure A(i: integer);
begin if i > 0 then
      begin D(i-1); x := x-h; plot;
            A(i-1); y := y-h; plot;
            A(i-1); x := x+h; plot;
            B(i-1)
      end
end ;
procedure B(i: integer);
begin if i > 0 then
      begin C(i-1); y := y+h; plot;
            B(i-1); x := x+h; plot;
            B(i-1); y := y-h; plot;
            A(i-1)
      end
end ;
```

```
procedure C(i: integer);
begin if i > 0 then
      begin B(i-1); x := x+h; plot;
            C(i-1); y := y+h; plot;
            C(i-1); x := x-h; plot;
            D(i-1)
      end
end ;
procedure D(i: integer);
begin if i > 0 then
      begin A(i-1); y := y-h; plot;
            D(i-1); x := x-h; plot;
            D(i-1); y := y+h; plot;
            C(i-1)
      end
end ;
begin startplot;
   i := 0; h := h0; x0 := h div 2; y0 := x0;
   repeat {plot Hilbert curve of order i}
      i := i+1; h := h div 2;
      x0 := x0 + (h div 2); y0 := y0 + (h div 2);
      x := x0; y := y0; setplot;
      A(i)
   until i = n;
   endplot
end
```

以上のものと似てはいるが，もうう少し複雑でもっと美的に洗練された例が**図3.7**に与えられている．このパターンはいくつかの曲線を重ね合わせたものであるが，そのうちの2つが**図3.6**に示されている．S_i は位数 i の Sierpinski 曲線と呼ばれている．

再帰の形はどのようになっているであろうか．基本構成単位は，葉 S_1 からその1つの辺を取り除いたものであると考えてしまうであろうが，これは正解ではない．Sierpinski 曲線と Hilbert 曲線の主な違いは，Sierpinski 曲線が（交点をもたない）閉曲線であることである．このことは，基本再帰形が開いた曲線でなければならず，また，4つの部分は再帰パターン自身には属さないリンクによって結合されていることを意味する．実際，これらのリンクは，最も外側の4つの“角”に属する4本の直線から成り，図3.6では太線で描かれてい

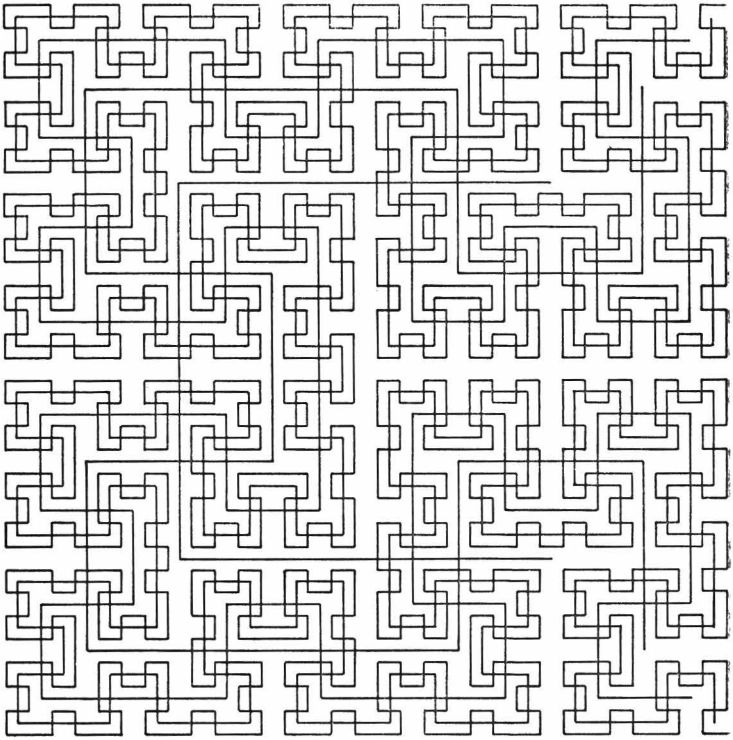


図 3.5　Hilbert 曲線H_1, …, H_5

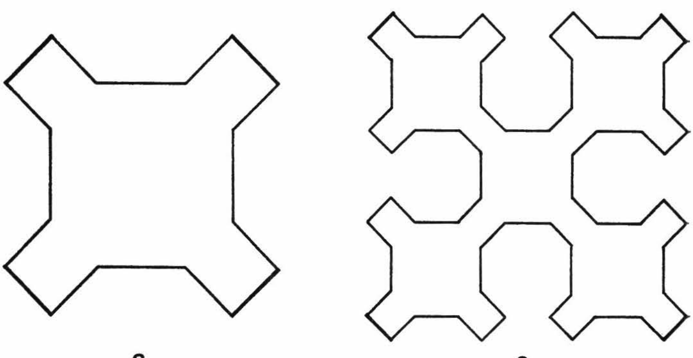


図 3.6　位数1および2の Sierpinski 曲線

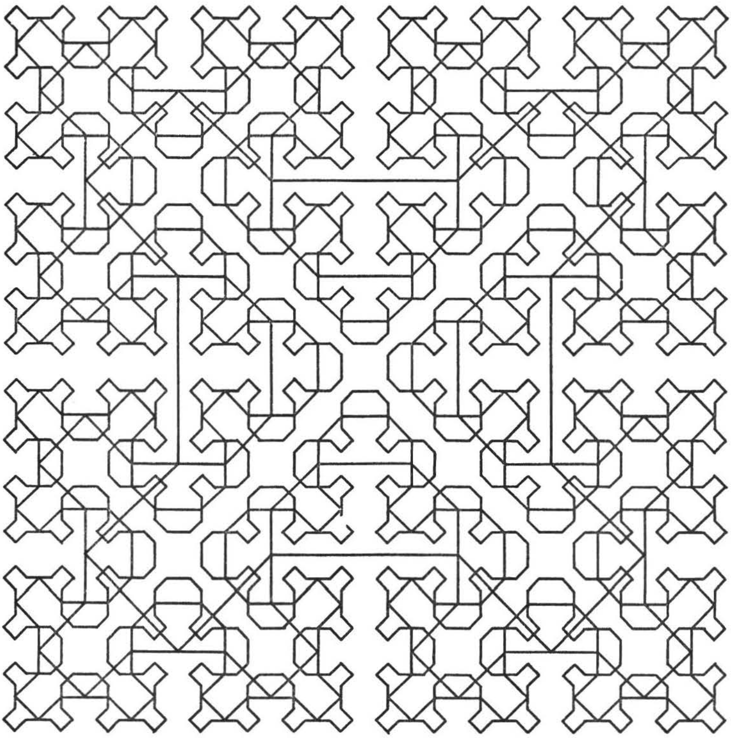


図 3.7 Sierpinski 曲線 $S_1, \ldots, S_4$

る．それらは，**空でない**初期曲線 S_0 に属していると考えられるが，この S_0 は1つの角で立っている正方形である．

さて，再帰の形は簡単に定められる．4つの部分パターンを再び A, B, C および D で表わし，また結合線もそのままの形で描くことにする．4つの再帰パターンは90度の回転を除いては実際には同一であることに注意してほしい．

Sierpinski 曲線の基本パターンは，

$$S: A \searrow B \swarrow C \nwarrow D \nearrow \qquad (3.21)$$

であり，再帰のパターンは，(3.22) のとおりである（2重矢印は2単位の長さの直線を表わす）．

$$
\begin{aligned}
A&: A \searrow B \Rightarrow D \nearrow A \\
B&: B \swarrow C \Downarrow A \searrow B \\
C&: C \nwarrow D \Leftarrow B \swarrow C \\
D&: D \nearrow A \Uparrow C \nwarrow D
\end{aligned}
\tag{3.22}
$$

プロット操作のための基本命令として，Hilbert 曲線の場合と同じものを用いるとすれば，上の再帰形は難なく次の（直接的および間接的に）再帰的なアルゴリズムに変換される．

```
procedure A(i: integer);
begin if i > 0 then
      begin A(i-1); x := x+h; y := y-h; plot;
            B(i-1); x := x+2*h;  plot;
            D(i-1); x := x+h; y := y+h; plot;
            A(i-1)
      end
end
```

(3.23)

この手続きは，再帰形（3.22）の第1行から導かれたものであり，パターン B, C および D に対応する手続きも同様に導かれる．主プログラムは，パターン（3.21）に従って構成される．その役目は，**プログラム 3.2** に示されるよう

プログラム 3.2　Sierpinski 曲線

```
program Sierpinski (pf,output);
{plot Sierpinski curves of orders 1 to n}
const n = 4; h0 = 512;
var i,h,x,y,x0,y0: integer;
   pf: file of integer;     {plot file}
procedure A(i: integer);
begin if i > 0 then
      begin A(i-1); x := x+h; y := y-h; plot;
            B(i-1); x := x + 2*h; plot;
            D(i-1); x := x+h; y := y+h; plot;
            A(i-1)
      end
end ;
```

```
procedure B(i: integer);
begin if i > 0 then
      begin B(i-1); x := x-h; y := y-h; plot;
            C(i-1); y := y - 2*h; plot;
            A(i-1); x := x+h; y := y-h; plot;
            B(i-1)
      end
end ;
procedure C(i: integer);
begin if i > 0 then
      begin C(i-1); x := x-h; y := y+h; plot;
            D(i-1); x := x - 2*h; plot;
            B(i-1); x := x-h; y := y-h; plot;
            C(i-1)
      end
end ;
procedure D(i: integer);
begin if i > 0 then
      begin D(i-1); x := x+h; y := y+h; plot;
            A(i-1); y := y + 2*h; plot;
            C(i-1); x := x-h; y := y+h; plot;
            D(i-1)
      end
end ;
begin startplot;
   i := 0; h := h0 div 4; x0 := 2*h; y0 := 3*h;
   repeat i := i+1; x0 := x0-h;
      h := h div 2; y0 := y0+h;
      x := x0; y := y0; setplot;
      A(i); x := x+h; y := y-h; plot;
      B(i); x := x-h; y := y-h; plot;
      C(i); x := x-h; y := y+h; plot;
      D(i); x := x+h; y := y+h; plot;
   until i = n;
   endplot
end
```

に，描画のための座標の初期設定および紙のサイズに対応して単位直線の長さ h を決定することである．$n=4$ としてこのプログラムを実行させた結果が図

3.7に示されている．ただし，S_0は描かれていない．

これらの例における再帰の使用は，明らかにエレガントであり，また説得力のあるものである．これらのプログラムの正しさは，その構造と構成パターンから容易に導かれる．そのうえ，形式 (3.5) に対応したレベル・パラメータ i の明確な使用により停止性が保証されるが，これは再帰の深さが n より大きくなれないからである．この再帰的な構成と対比した場合，再帰の明確な使用を避けた等価なプログラムは，極度に厄介で不明瞭なものである．読者は，[3-3]に示されているプログラムを理解することを試み，この主張をぜひ納得してほしい．

3.4　バックトラック・アルゴリズム

プログラミングに関する特に興味ある努力のひとつに，“一般問題解決”がある．これは特定の問題に対する解を見つけるために，ある定まった計算規則に従うのではなく，試行錯誤を用いようとするものである．この方法に共通したパターンは，試行錯誤の過程をいくつかの部分作業に分解していることである．多くの場合，これらの作業は再帰的に表現されるのが最も自然であり，そして，ある有限個の探索から構成される．一般的にはこの過程全体は，探索の木を徐々に構成し，走査する(枝刈りをする)ような試行・探索過程とみることができる．多くの問題では，与えられたパラメータによってはこの探索木は非常に急速に，そして通常は指数関数的に成長する．探索に要する努力もそれに応じて増加する．探索木は，しばしば，発見的方法のみを用いて枝刈りされ，それによって計算量を許容できる範囲内に減少させることができる．

一般的な発見的規則を論じることが，ここでのわれわれの目的ではない．むしろ，そのような問題解決の作業を部分作業に分解するための一般原理と，それへの再帰の応用が本章の主題である．まず初めに，良く知られた**騎士の巡回** (knight's tour) という例を使って，基本となる手法を示すことから始めよう．

n^2 個のます目からなる $n \times n$ のチェス盤が与えられているとする．ナイト(騎士)——チェスの規則に従って動くことが許された——が初期座標 x_0, y_0 のます目に置かれている．このとき問題は，もし可能ならば，盤全体をおおう，すなわち盤のます目がちょうど1回だけ訪問されるような n^2-1 回の移動からなる**巡回路**(tour)を計算することである．

n^2 個のます目をおおうというこの問題を簡単化する明らかな方法は，次の移動を行なう，あるいは，そのような移動が不可能なことを発見する問題につ

いて考えることである．したがって，次の移動を試みるための1つのアルゴリズムを定式化しよう．最初のアプローチは(3.24)に示されている．

procedure *try next move*;
begin *initialize selection of moves*;
　repeat *select next candidate from list of next moves*;
　　if *acceptable* **then**
　　begin *record move*;
　　　if *board not full* **then**
　　　begin *try next move*;　　　(3.24)
　　　　if *not successful* **then** *erase previous recording*
　　　end
　　end
　until (*move was successful*) $\vee$ (*no more candidates*)
end

このアルゴリズムをもっと精密に記述しようとするには，データ表現についていくつかの決定をしなければならない．明らかなステップは，盤をマトリックス，たとえば h，によって表現することである．また，インデックス値を表わす型も導入する．

type *index* $= 1..n$;
var h: **array**[*index*, *index*] **of** *integer*　　　(3.25)

盤のます目を表わすのに論理値の代わりに整数を用いたのは，連続したます目の占有の経過を記録したいからである．次のような明らかな規約を設ける．

$h[x, y] = 0$:　ます目 $\langle x, y\rangle$ は訪問されなかった
$h[x, y] = i$:　ます目 $\langle x, y\rangle$ は i 番目の移動で訪問された　　　(3.26)
　　$(1 \le i \le n^2)$

次の決定は適切なパラメータを選択することに関係がある．それらのパラメータは，次の移動のための初期条件を決定し，またその成功を記録するためのものである．前者のためには，移動を始める座標 x，y を指定し，移動の回数(記録のために)を指定すれば十分である．後者のためには，結果を表わす論理型のパラメータが必要である：$q = true$ は成功を，$q = false$ は失敗を表わす．

さて，これらの決定のもとに，どの文が詳細化できるであろうか．確かに，"*board not full*"（盤はいっぱいでない）は，"$i < n^2$"と表わせる．さらに，もし2つの局所的変数 u と v を導入して，ナイトの動き方の形から決まる可

能な移動先を表わせば，述語 "*acceptable*"（許される）は，新しいます目が盤上にある，すなわち $1 \leq u \leq n$ および $1 \leq v \leq n$ という条件と，以前に訪問したことがない，すなわち $h[u, v] = 0$ という条件の論理的結合として表現できる．合法的な移動を記録するという操作は，代入文 $h[u, v] := i$ によって，またこの記録の取り消しは，$h[u, v] := 0$ によって表わされる．もし局所的変数 $q1$ を導入して，このアルゴリズムの再帰的呼出しにおける結果のパラメータとして使えば，この $q1$ は "*move was successful*"（移動は成功であった）のところに代入できる．したがって，(3.27) に示される定式化が得られる．

```
procedure try (i: integer; x,y: index; var q: boolean);
var u,v: integer; q1: boolean;
begin initialize selection for moves;
      repeat let u,v be the coordinates of the next move defined
            by the rules of chess;
            if (1≤u≤n) ∧ (1≤v≤n) ∧ (h[u,v]=0) then
            begin h[u,v] := i;
                  if i < sqr(n) then                              (3.27)
                  begin try(i+1,u,v,q1);
                        if ¬q1 then h[u,v] := 0
                  end else q1 := true
            end
      until q1 ∨ (no more candidates);
      q := q1
end
```

あと1個所だけ詳細化を行なえば，われわれの基本的なプログラム記法によって完全に表現されたプログラムが得られる．これまではナイトの動き方の規則とは全く独立にプログラムが作られてきたことに注意してほしい．問題の特殊性を考慮することをこのように遅らせたのは，よくよく考えてのことであった．しかしながら，今やそれを考慮すべきときである．

出発点の座標対 $\langle x, y \rangle$ が与えられたとき，行先の座標 $\langle u, v \rangle$ の可能な候補としては8つある．それらには，**図3.8** のように1から8までの番号をつける．

x, y から u, v を得る簡単な方法は座標値の差を加えることであるが，この差は，差の対から成る1つの配列か，または1つの差から成る2つの配列に格納しておけばよい．
これらの配列を a と b で表わし，適当に初期設定しておく．このとき，"*next*

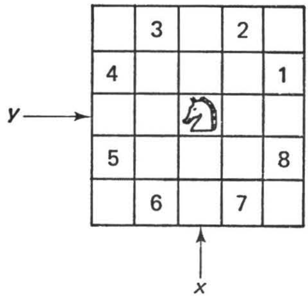


図 3.8 ナイトの 8 つの可能な動き方

candidate" (次の候補) を示すのにインデックス k を用いることができる．詳細は**プログラム** 3.3 に与えられている．再帰的手続きは，最初は巡回の出発点のます目の座標 x_0, y_0 をパラメータとして呼び出される．このます目には値 1 が与えられなければならず，また他のすべてのます目には非占有のマークをつけておかなければならない．

$$h[x_0, y_0] := 1; \quad try(2, x_0, y_0, q)$$

もう 1 つの詳細を見過ごしてはいけない：変数 $h[u, v]$ が存在するのは，u および v が配列限界 1, …, n 以内の場合だけである．したがって，(3.27) 中の式のうちで (3.24) 中の "*acceptable*" のところに代入されたものは，その初めの 2 つの成分項が真のときのみ有効なものである．プログラム 3.3 では，その適当な変形が示されている．さらにそこでは，2 重の関係 $1 \leq u \leq n$ が式 u **in** $[1, 2, \cdots, n]$ によって置き換えられているが，n が十分に小さい時には，この方がたぶん効率が良いであろう (1.10.3 節参照)．**表** 3.1 に，$n = 5$ かつ初期位置 ⟨1, 1⟩, ⟨3, 3⟩, および $n = 6$ かつ初期位置 ⟨1, 1⟩ の場合の解が示されている．

結果のパラメータ q と局所変数 $q1$ を 1 つの大域的変数で置き換えることが可能であり，これによりプログラムはいくらか簡単になる．

さて，この例題からどのような抽象化がなされ得るであろうか．そして，そ

プログラム 3.3 騎士の巡回

```
program knightstour (output);
const n = 5; nsq = 25;
type index = 1..n;
var i,j: index;
    q: boolean;
```

```
    s: set of index;
    a,b: array [1.8] of integer;
    h: array [index, index] of integer;
procedure try (i: integer; x,y: index; var q: boolean);
    var k,u,v: integer; q1: boolean;
begin k := 0;
    repeat k := k+1; q1 := false;
        u := x + a[k]; v := y + b[k];
        if (u in s)∧(v in s) then
        if h[u,v] = 0 then
        begin h[u,v] := i;
            if i < nsq then
                begin try (i+1,u,v,q1);
                    if ¬q1 then h[u,v] := 0
                end else q1 := true
        end
    until q1 ∨ (k=8);
    q := q1
end {try} ;
begin s := [1,2,3,4,5];
    a[1] :=  2; b[1] :=  1;
    a[2] :=  1; b[2] :=  2;
    a[3] := -1; b[3] :=  2;
    a[4] := -2; b[4] :=  1;
    a[5] := -2; b[5] := -1;
    a[6] := -1; b[6] := -2;
    a[7] :=  1; b[7] := -2;
    a[8] :=  2; b[8] := -1;
    for i := 1 to n do
        for j := 1 to n do h[i,j] := 0;
    h[1,1] := 1; try(2,1,1,q);
    if q then
        for i := 1 to n do
        begin for j := 1 to n do write(h[i,j]:5);
            writeln
        end
    else writeln(' NO SOLUTION ')
end
```

表 3.1　3つの騎士の巡回

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

23	10	15	4	25
16	5	24	9	14
11	22	1	18	3
6	17	20	13	8
21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

の抽象化はこの種の“問題解決”アルゴリズムに特有などのようなパターンを示しているのか．また，それはわれわれに何を教えてくれるのか．このアルゴリズムに特徴的な点は次の通りである．まず完全な解に至ると思われるステップが試みられ，記録される．そして，たぶんそれが完全な解には導かないステップであること，すなわち“行き止まり”へのステップであることが判明した時点で，そのステップまで戻って記録から消去される．この動作は**バックトラック**と呼ばれる．各ステップでの可能な候補の数を有限と仮定することにより，次の一般形(3.28)が(3.24)から導かれる．

```
procedure try;
begin intialize selection of candidates;
```

```
  repeat select next;
     if acceptable then
     begin record it;
        if solution incomplete then
        begin try next step;                                  (3.28)
           if not successful then cancel recording
        end
     end
  until successful ∨ no more candidates
end
```

もちろん実際のプログラムは形式(3.28)から導かれる色々な形を取り得る．よく出合うパターンは再帰の深さを示すレベル・パラメータを使用し，停止条件を単純化したものである．

さらに，各ステップで考察すべき候補の数がある一定の数，たとえば m，である場合には，次の形式 (3.29) が導かれる；これは文“*try*(1)”によって起動される．

```
procedure try(i: integer);
   var k: integer;
begin k := 0;
   repeat k := k+1; select k-th candidate;
      if acceptable then
      begin record it;
         if i < n then                                        (3.29)
         begin try(i+1);
            if not successful then cancel recording
         end
      end
   until successful ∨ (k = m)
end
```

本章の残りの部分ではさらに3つの例題を扱うが，これらは形式(3.29)の種々な具体化になっており，再帰の適切な使用例をさらに示すのがその目的である．

3.5　8王妃問題

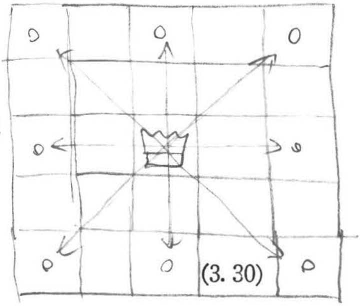
8王妃問題 (eight queen problem) は，試行錯誤やバックトラック・アルゴリズムの例題として良く知られている．この問題は1850年にC. F. Gauss に

よって研究されたが，彼はこれを完全には解決しなかった．しかしこれは驚くには当らない．なぜなら，解析的解法を受けつけないのがこの問題の特徴だからである．その代わり，それは多くの労力，忍耐および正確さを強要する．したがってそのようなアルゴリズムは，人間さらには天才などと比べても，より高度にそのような性格を持ち合わせた計算機によってのみ実行可能である．

8王妃問題は次のように述べられる（[3-4]も参照）：8個のクイーン(queen)を互いに他をとることができないようにチェス盤の上に配置せよ．

(3.29)の形式にしたがってプログラムを構成することにより，ただちに次のような解の初版を得る．

```
procedure try(i: integer);
begin
    initialize selection of positions for i-th queen;
    repeat make next selection;
        if safe then
        begin setqueen;
            if i < 8 then
            begin try(i+1);
                if not successful then remove queen
            end
        end
    until successful ∨ no more positions
end
```



(3.30)

先に進むためには，データ表現に関していくつかの考察が必要である．チェスの規則から，ひとつのクイーンは盤上でそれと同じ列，行および対角線上にあるすべてのコマをとることができるので，各列にはただ1つのクイーンしか置くことができず，よってi番目のクイーンはi番目の列に限定してもよいことがわかる．したがって，パラメータiは列のインデックスになり，位置(position)の選択(selection)の過程は，行インデックスjをその8個の可能な値のなかから選ぶことである．

盤上での8個のクイーンの表現の問題がまだ残っている．明らかな方法は，盤を表現するのに再び正方行列を用いることであるが，少し考えてみるとそのような表現をとった場合には位置の有効性の検査がかなりやっかいになることがわかる．一方，これは最も頻繁に実行される操作であるので，これでは非常に都合が悪い．したがって検査がなるべく簡単になるようなデータ表現を選ぶ

べきである．最良の方法は，真に適切で最も頻繁に利用される情報をできるだけ直接的に表現することである．われわれの場合には，これはクイーンの位置ではなく，各行および対角線上にクイーンが置かれているか否かということである．($1 \leq k \leq i$ に対して，各列 k にはただ1つのクイーンが置かれていることをわれわれはすでに知っている)．以上より次のような変数を導入する．

$$\begin{aligned}&\textbf{var}\ x : \textbf{array}\ [1\,..\,8]\ \textbf{of}\ \mathit{integer};\\ &\qquad a : \textbf{array}\ [1\,..\,8]\ \textbf{of}\ \mathit{boolean};\\ &\qquad b : \textbf{array}\ [b1\,..\,b2]\ \textbf{of}\ \mathit{boolean};\\ &\qquad c : \textbf{array}\ [c1\,..\,c2]\ \textbf{of}\ \mathit{boolean};\end{aligned} \tag{3.31}$$

ここで

$x[i]$ は i 番目の列のクイーンの位置を表わし，
$a[j]$ は j 番目の行にクイーンのいないことを意味し，
$b[k]$ は k 番目の↙対角線にクイーンのいないことを意味し，
$c[k]$ は k 番目の↘対角線にクイーンのいないことを意味する．

インデックスの上下限値 $b1$, $b2$, $c1$, $c2$ の選択は，インデックス b と c の計算のされ方に従って定められる：↙対角線上ではすべてのます目に対して，その座標 i と j の和は同じであり，また↘対角線上では座標差 $i-j$ は一定であることに注意しよう．適切な選択が**プログラム 3.4** に示されている．

これらのデータが与えられると，文 *setqueen* は次のように詳細化される．

$$x[i] := j;\ a[j] := \mathit{false};\ b[i+j] := \mathit{false};\ c[i-j] := \mathit{false} \tag{3.32}$$

また，*removequeen* は次のように詳細化される．

$$a[j] := \mathit{true};\ b[i+j] := \mathit{true};\ c[i-j] := \mathit{true} \tag{3.33}$$

そして条件 *safe* は，$\langle i, j\rangle$ がまだ空 (*true* で表現される) の行または対角線上にあることによって実現される．したがってこれは次の論理式によって表現される．

$$a[j] \wedge b[i+j] \wedge c[i-j] \tag{3.34}$$

以上でこのアルゴリズムの構成は終了するが，その全体はプログラム 3.4 に与えられている．計算された解は $x=(1, 5, 8, 6, 3, 7, 2, 4)$ であり**図 3.9** に示されている．

チェス盤から離れる前に，8王妃の例題を使って試行錯誤アルゴリズムのひ

とつの重要な拡張について述べよう．この拡張は，一般的な言葉でいえば，1つだけでなく，与えられた問題のすべての解を見つけるということである．

プログラム3.4　8　王　妃

```
program eightqueen1(output);
{find one solution to eight queens problem}
var i: integer; q: boolean;
    a: array [ 1.. 8] of boolean;
    b: array [ 2..16] of boolean;
    c: array [-7.. 7] of boolean;
    x: array [ 1.. 8] of integer;
procedure try(i: integer; var q: boolean);
    var j: integer;
begin j := 0;
    repeat j := j+1; q := false;
        if a[j] ∧b[i+j] ∧c[i-j] then
        begin x[i] := j;
            a[j] := false; b[i+j] := false; c[i-j] := false;
            if i < 8 then
            begin try (i+1,q);
                if ¬q then
                begin a[j] := true; b[i+j] := true; c[i-j] := true
                end
            end else q := true
        end
    until q ∨ (j=8)
end {try} ;
begin
    for i :=  1 to  8 do a[i] := true;
    for i :=  2 to 16 do b[i] := true;
    for i := -7 to  7 do c[i] := true;
    try (1,q);
    if q then
        for i := 1 to 8 do write (x[i]: 4);
    writeln
end
```

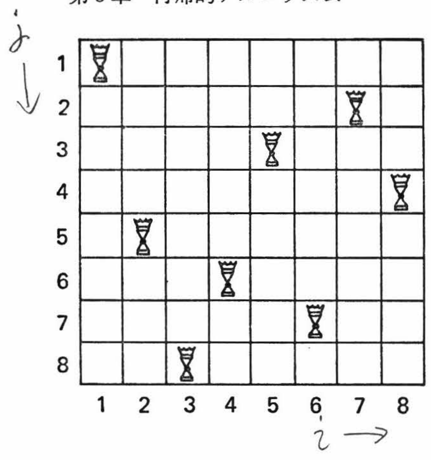


図3.9　8王妃問題の1つの解

この拡張は容易に受けいれることができる．座標の発生は系統的に行なっており，どの候補もたかだか1回しか発生されないことが保証されていることを思い出そう．アルゴリズムのこの性質は，候補木の探索が系統的に行なわれ，どの節点も正確に1回だけ訪問されることに対応する．したがって，1つの解が見つけられ，正しく記録されたならば，系統的な選択過程によって与えられた次の候補に進むだけでよいことになる．(3.29)から導かれる一般的形式が(3.35)に示されている．

```
procedure try(i: integer);
  var k: integer;
begin
  for k := 1 to m do
  begin select k-th candidate;
    if acceptable then
    begin record it;                                        (3.35)
      if i < n then try(i+1) else print solution;
      cancel recording
    end
  end
end
```

選択過程の停止条件を1つの項 $k = m$ と簡単化したことにより，repeat 文は適当な for 文によって置き換えられることに注意してほしい．すべての可能な解の探索が，1つの解の探索よりも簡単なプログラムによって実現されることは驚きである．

8王妃問題の92個のすべての解を決定するための拡張されたアルゴリズムが，**プログラム 3.5** に示されている．実際には，本質的に異なる解は12個しかないが，われわれのプログラムでは対称性を考慮していない．最初に発生された12個の解が**表 3.2** に示されている．右側の数 N は，ます目が *safe* かどうかの判定の実行頻度を示している．92個のすべての解についての平均値は161である．

プログラム 3.5　8　王　妃

```
program eightqueens (output);
var i: integer;
   a: array [ 1.. 8] of boolean;
   b: array [ 2..16] of boolean;
   c: array [-7.. 7] of boolean;
   x: array [ 1.. 8] of integer;
procedure print;
   var k: integer;
begin for k := 1 to 8 do write(x[k]: 4);
   writeln
end {print} ;
procedure try(i: integer);
   var j: integer;
begin
   for j := 1 to 8 do
      if a[j] ∧b[i+j] ∧c[i−j] then
      begin x[i] := j;
         a[j] := false; b[i+j] := false; c[i−j] := false;
         if i < 8 then try(i+1) else print;
         a[j] := true; b[i+j] := true; c[i−j] := true
      end
end {try} ;
begin
   for i :=  1 to  8 do a[i] := true;
   for i :=  2 to 16 do b[i] := true;
   for i := −7 to  7 do c[i] := true;
   try (1)
end
```

表 3.2　8王妃問題の12の解

X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	N
1	5	8	6	3	7	2	4	876
1	6	8	3	7	4	2	5	264
1	7	4	6	8	2	5	3	200
1	7	5	8	2	4	6	3	136
2	4	6	8	3	1	7	5	504
2	5	7	1	3	8	6	4	400
2	5	7	4	1	8	6	3	72
2	6	1	7	4	8	3	5	280
2	6	8	3	1	4	7	5	240
2	7	3	6	8	5	1	4	264
2	7	5	8	1	4	6	3	160
2	8	6	1	3	5	7	4	336

3.6　安定な結婚問題

等しい基数 n をもった共通部分のない2つの集合 A, B が与えられたと仮定しよう．このとき，ある**制約条件**を満たす n 個の対 $\langle a, b \rangle$，ただし a **in** A かつ b **in** B, からなる集合を見つける問題を考える．そのような対に対する基準としては多くの異なったものがあるが，それらの中に，"安定な結婚規則"と呼ばれる規則がある．

A は男性の集合，B は女性の集合としよう．各男性と女性は，自分の結婚相手としての好き嫌いをはっきり述べている．もし，n 個の組(カップル)を作ったとき，自分の実際の結婚相手よりお互いに好きであるような（結婚していない）男女が存在するならば，このような割付けは**不安定**であるといわれる．もしそのような組が存在しなければ，**安定** (stable) であるといわれる．

この状況は，好き嫌いにもとづいて割付けを行なわなければならないような多くの関連した問題，たとえば生徒による学校の選択，軍隊の各部門による新兵の選択，等を特徴づけている．結婚についてのこの例題は特に直感的である：しかしながら，好き嫌いの表明が不変であり，特定の割付けを行なった後でもそれが変らないことに注意せよ．この規則は問題を単純にはしているが，現実をゆがめている(抽象化と呼ばれる)．

解を見つける1つの方法は，2つの集合が空になるまでその要素を次々と取り出し，対を作ることを試みることである．**すべて**の安定な割付けを見つけるとしたとき，(3.35)の形式を用いた1つの解をただちに書き下すことができる．

try(*m*)は，男性 *m* に対する相手を見つけるアルゴリズムを示すとし，また，この探索はこの男性の好みの表明リストの順に行なわれるとする．これらの仮定にもとづいたアルゴリズムの初版は(3.36)で与えられる．

```
procedure try(m: man);
   var r: rank;
begin
   for r := 1 to n do
   begin pick the r-th preference of man m;
      if acceptable then
      begin record the marriage;
         if m is not last man then try(succ(m))
            else record the stable set;
         cancel the marriage
      end
   end
end
```
(3.36)

再び，データ表現についてある決定をしなければ先に進めない時点にわれわれは到達した．3つのスカラ型を導入するが，簡単のためにそれらの値は1から *n* までの整数とする．これらの3つの型は形式的には同一であるが，それらに異なった名前をつけることにより明快さがかなり増加する．特に，ある変数が何を表わしているかがより容易に判明するようになる．

```
type man   = 1..n;
     woman = 1..n;
     rank  = 1..n
```
(3.37)

初期データは，男性および女性の好き嫌いを表わす2つの配列によって表現される．

```
var wmr: array[man, rank] of woman
    mwr: array[woman, rank] of man
```
(3.38)

ここで，*wmr*[*m*] は男性 *m* の好みのリストを表わす，すなわち *wmr*[*m*][*r*] = *wmr*[*m*, *r*]は，男性 *m* のリストの中で *r* 番目を占める女性である．同様に，*mwr*[*w*] は女性 *w* の好みのリストであり，*mwr*[*w*, *r*] は彼女の *r* 番目の選択を表わす．

結果は女性からなる配列 *x* によって表わされるが，*x*[*m*] は男性 *m* の相手

表3.3　安定な結婚問題に対する入力データの例

順　　位	1	2	3	4	5	6	7	8
男性1が選ぶ女性	7	2	6	5	1	3	8	4
2	4	3	2	6	8	1	7	5
3	3	2	4	1	8	5	7	6
4	3	8	4	2	5	6	7	1
5	8	3	4	5	6	1	7	2
6	8	7	5	2	4	3	1	6
7	2	4	6	3	1	7	5	8
8	6	1	4	2	7	5	3	8
女性1が選ぶ男性	4	6	2	5	8	1	3	7
2	8	5	3	1	6	7	4	2
3	6	8	1	2	3	4	7	5
4	3	2	4	7	6	8	5	1
5	6	3	1	4	5	7	2	8
6	2	1	3	8	7	4	6	5
7	3	5	7	2	4	1	8	6
8	7	2	8	4	5	6	3	1

を表わす．男性と女性の間の対称性——“男女同権”とも呼ばれる——を保つために付加的な配列 y を導入するが，$y[w]$は女性の相手を表わす．

$$\begin{aligned}\textbf{var}\ x\text{: }\textbf{array}[\mathit{man}]\ \textbf{of}\ \mathit{woman};\\ y\text{: }\textbf{array}[\mathit{woman}]\ \textbf{of}\ \mathit{man}\end{aligned} \tag{3.39}$$

y は x にある情報しか持っていないので，y がどうしても必要というわけではないことは明白である．事実，関係

$$x[y[w]]=w,\quad y[x[m]]=m \tag{3.40}$$

が結婚しているすべての m, w について成立する．したがって $y[w]$ の値は x の単純探索によって決定できる．しかしながら，配列 y は明らかにアルゴリズムの効率を向上させる．x と y によって表現される情報は，提案された結婚の集合の安定性を判定するために必要である．この集合は，個々の男女を結婚させてはその提案された結婚の安定性を判定することにより徐々に構成されるので，x と y はそのすべての成分が定義される以前にも必要となる．定義された成分を記録するために論理配列を導入する．

$$\mathit{singlem}: \textbf{array}[\mathit{man}]\ \textbf{of}\ \mathit{boolean} \tag{3.41}$$

$$singlew : \textbf{array}[woman] \textbf{ of } boolean$$

これは次のような意味を表わす.

$\neg singlem[m]$ ならば $x[m]$ は定義されている.
$\neg singlew[w]$ ならば $y[w]$ は定義されている.

しかしながら, 提案されたアルゴリズムを調べてみると, 男性が結婚していることは簡単な方法, すなわち

$$\neg singlem[k] \equiv k < m \tag{3.42}$$

によって m の値から決定できることがすぐに明らかとなる. これより配列 *singlem* は省略してもよいことがわかり, したがってわれわれは名前 *singlem* を *single* と簡単化する.

これらの規約のもとに, (3.43) に示される詳細化が得られる. 述語 *acceptable* は *single* と *stable* の連言に詳細化されているが, *stable* はこれ以後さらに詳細化されるべき関数である.

```
procedure try(m: man);
   var r: rank;  w: woman;
begin for r := 1 to n do
   begin w := wmr[m,r];
      if single[w] ∧ stable then
      begin x[m] := w;  y[w] := m;  single[w] := false;      (3.43)
         if m < n then try(succ(m))
            else record stable set;
         single[w] := true
      end
   end
end
```

この時点では, この解とプログラム 3.5 との間の強い類似性はまだ顕著である.

さて決定的な作業は, 安定性を判定するアルゴリズムの詳細化である. 不幸にして, プログラム 3.5 におけるクイーンの位置の安全性のような単純な表現によっては, 安定性を表現することは不可能である. 考慮すべき第一の項目は, 安定性が好き嫌いあるいは順位の比較によって定義されているということである. しかしながら, 男性や女性の順位は, これまで設定してきたデータの集合中にはどこにも陽な形では存在していない. 確かに男性 m の意中の女性 w の順位は計算することはできるが, それは $wmr[m]$ 中での w の高価な探索によ

ってのみ可能なのである.

安定性の計算は非常に頻繁に行なわれる操作であるので，この情報がもっと直接的にアクセス可能であるようにしておくことは重要である．このために2つの行列を導入する.

$$\begin{aligned} &rmw : \textbf{array}\,[man, woman]\ \textbf{of}\ rank; \\ &rwm : \textbf{array}\,[woman, man]\ \textbf{of}\ rank \end{aligned} \tag{3.44}$$

ここで，$rmw[m, w]$ は男性 m の好みのリスト中における女性 w の順位を表わし，$rwm[w, m]$ は w のリスト中の男性 m の順位を表わす．この補助配列の値は定数であり，wmr と mwr の値から初めに決定できることは明白である.

述語 *stable* を決定する過程は，今やその原定義に厳密に沿って進められる. m と $w = wmr[m, r]$，すなわち m の好みのリスト中の順位 r の w，との結婚の可能性を調べようとしていることを思い出してほしい．楽観的には，これまでは安定であったと仮定してトラブルの可能な原因を見つけることを試みる．それはどこに隠されている可能性があるであろうか．2つの対称的な可能性がある：

1. m は w より女性 pw が好きで，かつ彼女自身もその夫より m が好きであるような女性 pw が存在するかも知れない.
2. w は m より男性 pm が好きで，かつ彼自身もその妻より w が好きであるような男性 pm が存在するかも知れない.

トラブルの原因1を追求するために，m が w より好きであるすべての女性 pw について，すなわち $i < r$ であるすべての $pw = wmr[m, i]$ に対して，順位 $rwm\,[pw, m]$ と $rwm\,[pw, y\,[pw]]$ を比較する．偶然にも，候補となる女性はすべて既に結婚していることがわかる．なぜなら，もし彼女等のうちのだれかが未婚であったならば，m は以前に彼女を選んでいたはずだからである．今述べた過程は単純な線形探索によって定式化することができる：s は安定性を表わす.

```
s := true;  i := 1;
while (i < r) ∧ s do
   begin pw := wmr[m,i];  i := i+1;                          (3.45)
      if ¬single[pw] then s := rwm[pw, m] > rwm[pw, y[pw]]
   end
```

トラブルの原因2を追求するには，w が現在の相手 m より好きなすべての候

プログラム 3.6 安定な結婚

```
program marriage (input,output);
{problem of the stable marriages}
const n = 8;
type man = 1..n; woman = 1..n; rank = 1..n;
var m: man; w: woman; r: rank;
    wmr: array [man, rank] of woman;
    mwr: array [woman, rank] of man;
    rmw: array [man, woman] of rank;
    rwm: array [woman, man] of rank;
    x:   array [man] of woman;
    y:   array [woman] of man;
    single: array [woman] of boolean;

procedure print;
    var m: man; rm, rw: integer;
begin rm := 0; rw := 0;
    for m := 1 to n do
    begin write (x[m]:4);
        rm := rm + rmw[m,x[m]]; rw := rw + rwm[x[m],m]
    end ;
    writeln (rm:8,rw:4);
end {print} ;

procedure try(m: man);
    var r: rank; w: woman;

    function stable: boolean;
        var pm: man; pw: woman;
            i, lim: rank; s: boolean;
    begin s := true; i := 1;
        while (i<r) ∧ s do
        begin pw := wmr[m,i]; i := i+1;
            if ¬single[pw] then s := rwm[pw,m] > rwm[pw,y[pw]]
        end ;
        i := 1; lim := rwm[w,m];
        while (i<lim) ∧ s do
        begin pm := mwr[w,i]; i := i+1;
            if pm < m then s := rmw[pm,w] > rmw[pm,x[pm]]
        end ;
```

```
        stable := s
    end {stable} ;
begin {try}
    for r := 1 to n do
    begin w := wmr[m,r];
        if single[w] then
            if stable then
            begin x[m] := w; y[w] := m; single[w] := false;
                if m < n then try(succ(m)) else print;
                single[w] := true
            end
    end
end {try} ;
begin {main program}
    for m := 1 to n do
        for r := 1 to n do
        begin read(wmr[m,r]); rmw[m,wmr[m,r]] := r
        end ;
    for w := 1 to n do
        for r := 1 to n do
        begin read(mwr[w,r]); rwm[w,mwr[w,r]] := r
        end ;
    for w := 1 to n do single[w] := true;
    try (1)
end
```

補 pm について調べなければならない．すなわち，調査対象は，$i < rwm[w, m]$ であるようなすべての男性 $pm = mwr[w, i]$ である．トラブルの原因1の調査と同様に，順位 $rmw[pm, w]$ と $rmw[pm, x[pm]]$ の比較が必要になるであろう．しかしながら，いまだ独身である pm に対する $x[pm]$ を含んだ比較を除外することに注意しなければならない．このためには判定 $pm < m$ を行なえばよい．なぜなら，m より前のすべての男性はすでに結婚しているからである．

完全なアルゴリズムが**プログラム 3.6** に示されている．**表 3.3** は配列 wmr と mwr を表わしている入力データ集合である．最後に，計算された9個の安定な解が**表 3.4** に与えられている．

明らかにこのアルゴリズムは単純なバックトラック形式に基づいたものであ

表 3.4 安定な結婚問題の結果

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	rm	rw	c†
解 1	7	4	3	8	1	5	2	6	16	32	21
2	2	4	3	8	1	5	7	6	22	27	449
3	2	4	3	1	7	5	8	6	31	20	59
4	6	4	3	8	1	5	7	2	26	22	62
5	6	4	3	1	7	5	8	2	35	15	47
6	6	3	4	8	1	5	7	2	29	20	143
7	6	3	4	1	7	5	8	2	38	13	47
8	3	6	4	8	1	5	7	2	34	18	758
9	3	6	4	1	7	5	8	2	43	11	34

† 安定性判定の回数
解 1 = 男性にとって最適な解
解 9 = 女性にとって最適な解

る．その効率は，基本的には解の木の枝刈り法の工夫に依存している．いくらか高速ではあるが，さらに複雑で見通しの悪いアルゴリズムが McVitie と Wilson [3-1 および 3-2] によって発表されているが，彼等はまたこの問題を等しくない大きさの（男性および女性の）集合の場合に拡張した．

先の 2 つの例題のようなアルゴリズムは，（与えられた制約条件のもとで）ある問題に対するすべての可能な解を生成するものであるが，これらは解のなかからある意味で最適な 1 つまたは複数のものを選択するのによく用いられる．現在の例題についていえば，たとえば，男性――あるいは女性――またはすべての人を平均的に最も良く満足させるような解に興味があるかも知れない．

表 3.4 には，すべての女性について，彼女等の夫の好みのリストにおける彼女等自身の順位の和，およびすべての男性について，彼等の妻の好みのリストにおける彼等自身の順位の和が示されていることに注意してほしい．これらの値は次式によって計算されたものである．

$$rm=\sum_{m=1}^{n} rmw[m, x[m]], \qquad rw=\sum_{m=1}^{n} rwm[x[m], m] \tag{3.46}$$

値 rm が最小な解は男性にとって最適な安定解と呼ばれ，rw が最小な解は女性にとって最適な安定解である．男性にとって良い解が最初に生成され，女性にとって良い解が終わりの方に現われているのは，用いられた探索方針の性質によるものである．この意味でこのアルゴリズムは男性側にかたよったものである．これは男性と女性の役割を系統的に交換する，すなわち mwr と wmr

および *rmw* と *rwm* とを交換することにより，ただちに変更することができる.

ここでは，このプログラムをこれ以上拡張するのはやめて，最適な解の探索の組込みは，次の，そしてバックトラック・アルゴリズムの最後の例題として残しておく.

3.7　最適選択問題

バックトラック・アルゴリズムの最後の例題は，一般形(3.35)で表現される前の2つの例題の論理的な拡張である．最初われわれは与えられた問題に対する**1つ**の解を見つけるためにバックトラックの原理を用いた．これには騎士の巡回と8王妃問題を例題に用いた．次に，与えられた問題の**すべて**の解を求めるという目標に取り組んだ．用いられた例題は，8王妃および安定な結婚問題であった．そしてここでは，**最適**な解を見つける問題を考えよう.

このためには，すべての解を生成し，そしてそれらを生成する過程において，ある意味で最適なものを1つ保持しておくことが必要である．最適性がある正値関数 $f(s)$ で定義されていると仮定すれば，一般形 (3.35) において，文 *print solution* (解を印刷せよ) を次の文によって置き換えることにより求めるアルゴリズムが導かれる.

$$\textbf{if}\ f(solution) > f(optimum)\ \textbf{then}\ optimum := solution \tag{3.47}$$

変数 *optimum* には，これまでに出合った最良の解を記録しておく．もちろんそれは適当に初期設定されなければならない．そのうえ $f(optimum)$ の値は頻繁な再計算を避けるために，他の変数に記録しておくのが普通である.

与えられた問題に対する最適な解を見つけるという一般的問題の例として，ある制約条件のもとで，与えられたもの (object) の集合から**最適な選択**を見つける問題がある．容認できる (acceptable) 解を構成する選択は，基本集合中の個々のものを調べることにより徐々に作り上げられる．手続き *try* は個々のものの適合性を調べるプロセスを記述しており，すべてのものを調査し終わるまで (次のものを調べるために) 再帰的に呼び出される.

個々のもの (前の例では候補と呼ばれた) の調査には，2つの結果があり得る．すなわちそのものを現在の選択に**含める** (inclusion) か，あるいは**除外する** (exclusion) かである．これは **repeat** または **for** 文の使用を不適当なものにする．その代わり，2つの場合を陽に書き分ければよい．これは (3.48) に示されている (ものは $1, 2, \cdots, n$ と番号づけられていると仮定する).

```
procedure try(i: integer);
begin
1: if inclusion is acceptable then
   begin include i-th object;
      if i < n then try(i+1) else check optimality;
      eliminate i-th object                                  (3.48)
   end;
2: if exclusion is acceptable then
         if i < n then try(i+1) else check optimality
end
```

このパターンから，2^n の可能な集合があることは明らかである．したがって適当な容認基準を用いて，調べるべき候補の数を大巾に減らさなければならない．このプロセスを明瞭にするために，選択問題として1つの具体的問題を選ぼう：n 個のもの a_1, …, a_n のどれもその重さ w と価値 v によって特徴づけられていると仮定しよう．そして最適な集合とは，その要素の価値の和が最大であるものとする．ただし，それらの重さの和に対する制限が制約条件として与えられているとする．これは，n 個の品物のうちから適当に選んでスーツケースに詰め，それらの価値の総和を最適にし，かつそれらの全重量を特定の許容値以下にしなければならない旅行者なら誰でも良く知っている問題である．

さて，与えられた事実のデータによる表現法を決定しよう．(3.49) の決定はこれまでの展開から容易に導かれる．

```
type index = 1..n;
     object = record w,v: integer end
var a: array [index] of object;                              (3.49)
    limw, totv, maxv: integer;
    s, opts: set of index
```

変数 *limw* と *totv* は制限重量および n 個のものの価値の総和を表わす．これらの値は，選択過程全体を通して実際には定数である．*s* は現在の選択を表わすが，そこでは個々のものは名前(インデックス)によって表わされている．*opts* はこれまでに出合ったうちの最適な選択を表わし，*maxv* はその値である．

あるものが現在の選択として容認可能なための基準は何であろうか．*inclusion* (包含) について考える時には，それが重量制限に適合する場合には選択可能である．もし適合しないならば，現在の選択にこれ以上ものを追加するこ

とを止めてよいであろう．しかしながら，*exclusion*（除外）について考えるときには，容認可能性，すなわち現在の選択をそのまま拡大してよいための基準は，この除外以後に達成可能な価値の総和がこれまでの最適値より小さくならないことである．なぜなら，もし小さくなったならば，この探索の継続はある解を作り出すには違いないが，最適解を生み出すことはないからである．したがって現在のパスについてのこれ以上の探索は無駄である．これらの条件から，選択プロセスの各段階で計算しなければならない量が決定される．

1. これまでの選択 s の総重量 tw
2. 現在の選択からまだ達成できる値 av

これら2つの値は，手続き *try* のパラメータとして表現するのが適当である．
(3.48)の条件 *inclusion is acceptable*（包含は容認される）は

$$tw + a[i].w \le limw \tag{3.50}$$

のように定式化され，それに続く *check optimality*（最適性を検査する）は次のようになる．

```
if av > maxv then
   begin {new optimum, record it}
      opts := s;   maxv := av
   end
```
(3.51)

最後の代入文は，n 個のものがすべて処理された後では，達成可能な値は達成された値に等しいという論法に基づいている．
(3.48)の条件 *exclusion is acceptable*（除外は容認される）は

$$av - a[i].v > maxv \tag{3.52}$$

によって表現される．これはこれ以後再び使われるので，$av - a[i].v$ には名前 $av1$ を与えて再計算を行なわないようにする．

したがって，大域変数のための適切な初期設定文を付け加えると，全プログラムは(3.48)から(3.52)により次のようになる．集合 s に対する包含と除外が集合演算により容易に表現されることは注目に値する．重量制限値が10から120の間の場合について，**プログラム3.7**の実行結果が**表3.5**に与えられている．探索木の成長を妨げるための制限条件を伴なったこのバックトラック・アルゴリズムは，**分岐限定アルゴリズム**（branch and bound algorithm）として知られている．

プログラム 3.7 最適選択

```
program selection (input,output);
{find optimal selection of objects under constraint}
const n = 10;
type index = 1 .. n;
   object = record v,w: integer end;
var i: index;
   a: array [index] of object;
   limw, totv, maxv: integer;
   w1, w2, w3: integer;
   s, opts: set of index;
   z: array [boolean] of char;

procedure try(i: index; tw,av: integer);
   var av1: integer;
begin {try inclusion of object i}
   if tw + a[i] .w ≤ limw then
   begin s := s + [i];
      if i < n then try(i+1, tw+a[i].w, av) else
         if av > maxv then
         begin maxv := av; opts := s
         end ;
      s := s − [i]
   end ;
   {try exclusion of object i}  av1 := av − a[i] .v;
   if av1 > maxv then
   begin if i < n then try(i+1, tw, av1) else
            begin maxv := av1; opts := s
            end
   end
end {try} ;

begin totv := 0;
   for i := 1 to n do
      with a[i] do
      begin read(w,v); totv := totv + v
      end ;
   read(w1,w2,w3);
   z[true] := '*'; z[false] := ' ';
   write(' WEIGHT    ');
   for i := 1 to n do write (a[i] .w: 4);
```

```
  writeln; write ('  VALUE       ');
  for i := 1 to n do write (a[i] .v: 4);
  writeln;
  repeat limw := w1; maxv := 0; s := [ ]; opts := [ ];
     try(1,0,totv);
     write(limw);
     for i := 1 to n do write('       ', z[i in opts]);
     writeln; w1 := w1 + w2
  until w1 > w3
end
```

表 3.5　最適選択問題からの出力例

重　さ	10	11	12	13	14	15	16	17	18	19
価　値	18	20	17	19	25	21	27	23	25	24
10	*									
20							*			
30					*		*			
40	*				*		*			
50	*	*		*			*			
60	*	*	*	*	*					
70	*	*			*		*		*	
80	*	*	*		*		*	*		
90	*	*			*		*		*	*
100	*	*		*	*		*	*	*	
110	*	*	*	*	*	*	*		*	
120	*	*			*	*	*	*	*	*

演 習 問 題

3.1　(ハノイの塔) 3本の棒と大きさの異なる n 枚の円盤がある．円盤には穴があいていて，棒にさして積み重ねることができ，“塔”を構成することができる．最初 n 枚の円盤は大きさの順に棒 A のところに置かれている．$n=3$ の場合が**図 3.10** に示されている．このとき，棒 A から C に n 枚の円盤をその順序を変えないように移せ．ただし，次の制約条件のもとで行なえ．

1. 各ステップではただ一枚の円盤のみが移動可能である．
2. どの円盤もそれより小さい円盤の上に置いてはいけない．

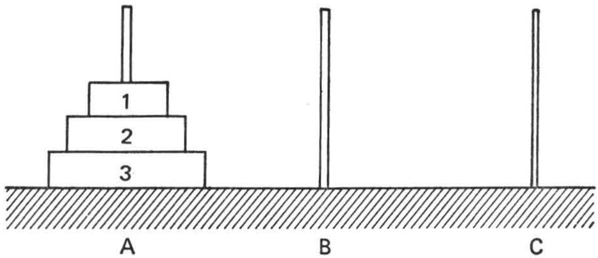


図 3.10 ハノイの塔

3. 棒 B は補助的な場所として使用してよい．

これを行なうためのアルゴリズムを見つけよ．1つの塔は，その最上の1枚の円盤と，残りの円盤からなる塔から構成されている，と考えるのが都合がよいことに注意せよ．このアルゴリズムを再帰的プログラムとして書け．

3.2 n 個の要素 a_1，…，a_n の $n!$ 個のすべての順列をその場所で，すなわち他の配列を用いずに，生成する手続きを書け．1つの順列を生成するごとに補助的手続 Q を呼び出して，たとえば生成された順列を出力するものとする．

ヒント：要素 a_1，…，a_m のすべての順列を生成するという作業は，a_1，…，a_{m-1} のすべての順列を生成し，その後で a_m を出力するという m 個の部分作業から構成されると考えよ．ただし，i 番目の部分作業では，2つの要素 a_i と a_m が初めから交換されていたとする．

3.3 4つの曲線 W_1, W_2, W_3, W_4 の重ね合わせである**図 3.11** の再帰形式を推定せよ．この構造は Sierpinski 曲線 (3.21) と (3.22) のそれに似ている．再帰パターンよりこの曲線を描く再帰的プログラムを導け．

3.4 8王妃プログラム 3.5 で計算される 92 個の解のうち，12 個のものだけが本質的に異なるものである．他のものは，軸あるいは中心点に関する対称として導くことができる．12 個の基本解を決定せよ．このとき，たとえば列1の探索は位置 1～4 に限定してもよいことに注意せよ．

3.5 安定な結婚プログラムを変更して，最適解（男性あるいは女性にとっての）を決定するようにせよ．したがって，それはプログラム 3.7 で表わされた形の“分岐限定”プログラムになる．

3.6 ある鉄道会社では n 個の駅 S_1，…，S_n を開設しているが，計算機化

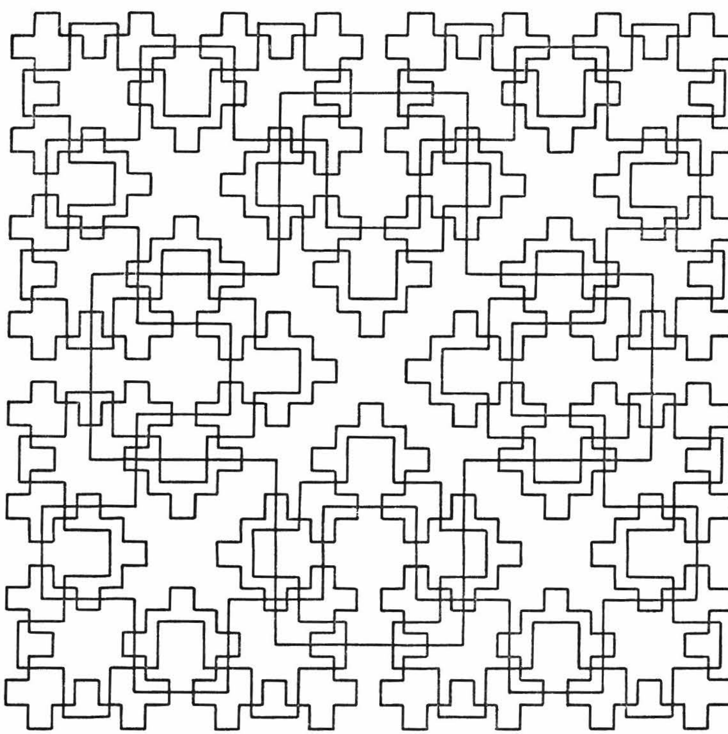


図 3.11　位数1から4のW-曲線

された情報端末によって，顧客情報サービスを改善したいと考えている．客が出発駅 S_A と目的駅 S_D を入力すると，（直ちに）旅程の全時間が最少になるような列車接続のスケジュールが与えられるようにしたい．

所要の情報を計算するためのプログラムを作れ．（諸君のデータ・バンクである）時間表は適切なデータ構造によって与えられているとし，それにはすべての利用可能な列車の出発（＝到着）時間が含まれているとする．もちろんすべての駅が直通線によって結ばれているとは限らない（演習問題1.8も参照せよ）．

3.7　すべての非負整数引数 m, n に対して，Ackermann 関数 A は次のように定義されている．

$$A(0, n) = n+1$$
$$A(m, 0) = A(m-1, 1) \qquad (m > 0)$$
$$A(m, n) = A(m-1, A(m, n-1)) \qquad (m, n > 0)$$

再帰を使用せずに $A(m, n)$ を計算するプログラムを設計せよ.

クイック・ソートの非再帰版であるプログラム 2.11 と同じ方針に従え.
一般に再帰的プログラムを繰返し型プログラムに変換するための規則を考案せよ.

参 考 文 献

3-1. McVitie, D. G. and Wilson, L. B., "The Stable Marriage Problem," *Comm. ACM*, **14**, No. 7 (1971), 486–92.

3-2. _____ _____, "Stable Marriage Assignment for Unequal Sets," *BIT*, **10**, (1970), 295–309.

3-3. "Space Filling Curves, or How to Waste Time on a Plotter," *Software-Practice and Experience*, **1**, No. 4 (1971), 403–40.

3-4. Wirth, N., "Program Development by Stepwise Refinement," *Comm. ACM*, **14**, No. 4 (1971), 221–27.

第4章

動的データ構造

4.1　再帰的データ型

第2章では，配列，レコードおよび集合構造が基本的データ型として導入された．それが“基本的”と呼ばれる理由は，より複雑な構造を形成する際の構成単位であり，かつ現実にそれらは最も頻繁に用いられるからである．あるデータ型を定義する，したがって，いくつかの変数がそのデータ型であることを指定する目的は，これらの変数の値域やその記憶パターンが一定に固定されることである．ゆえに，このように宣言される変数は**静的**と呼ばれる．しかしながら，より複雑な情報構造が必要となる多くの問題がある．これらの問題に共通なことは，それらの構造が計算の過程において変化することであり，それゆえに，それらは**動的**構造と呼ばれる．もちろんそのような構造の成分もその細部のあるレベルでは静的であり基本的データ型である．本章では動的情報構造の構成，解析および管理について述べる．

アルゴリズムの構造化の方法とデータの構造化の方法に緊密な類似性があることは良く知られている．すべての類似性には差異がつきものではあるが（そうでないと同一になってしまう），データとプログラムの構造化方法の類似性は際立っている．

初等的かつ構造のない文は代入文である．これはデータ構造の世界では構造のない型スカラに対応する．最も簡単な構造は，数え上げ，あるいは順序づけによって得られ，複合文とレコード構造がある．どちらも有限個の（通常は少数の）陽に数え上げられた成分からなるが，それらの成分は互いにすべて異なっていてもよい．もしすべての成分が同一であれば，それらは個々に書き下す必要はない：われわれは，ある有限の重複度を示すために **for** 文と **array** 構造を用いる．2つ以上のものの間の選択は，条件文あるいは **case** 文，および

可変レコード構造によって表現される．最後に，実行前には不確定な（そして潜在的には無限の）回数の繰返しは，**while** または **repeat** 文によって表現されるが，これに対応するデータ構造は，列(ファイル)である．これは無限に大きい基数をもった型のうちで最も簡単なものである．

手続き文についても同様に対応するデータ構造があるかどうかという疑問が生じる．手続きに関する性質のうちで，この点に関して最も興味がありかつ良く知られていないものは**再帰**である．そのような再帰的データ型の値は，自分自身に対する呼出しを1回以上含むような手続きからの類推によって，それ自身と同じデータ型に属する1個以上の成分を含むであろう．手続きの場合と同様に，そのようなデータ型は直接的にだけではなく，間接的にも再帰的になり得るであろう．

再帰的データ型によって表現するのが最も適当なものの簡単な例として，プログラム言語における算術式がある．この場合，再帰は入れ子すなわち式中のオペランドとして括弧で囲んで部分式の使用を可能にするために用いられる．したがって，ここで次のように（非形式的に）式を定義しよう．

> **式**は項の後に演算子を書き，その後に項を書いたものである（2つの項はその演算子のオペランドを構成する）．**項**は変数(識別名で表現される）であるか，または式を括弧で囲んだものである．

その値がこのような式を表わすデータ型は，すでにある道具と再帰を用いて容易に記述することができる†：

```
type expression = record op: operator;
                         opd1, opd2: term
                  end;
type term = record                                    (4.1)
                if t then (id: alfa)
                     else (subex: expression)
            end
```

ゆえに型 *term* のどの変数も2つの成分，すなわちタグ・フィールド *t* と，*t* が *true* であればフィールド *id*，また，もし *t* が *false* であればフィールド *subex* をもつ．

さて，例として次の4つの式を考えよう．

† **訳注** 標準的な PASCAL 文法では，*record* 中に **if** 構造がくることはないが，その意味は **case** 構造から容易に類推できるであろう，

$$\begin{aligned}&1.\ x+y\\&2.\ x-(y*z)\\&3.\ (x+y)*(z-w)\\&4.\ (x/(y+z))*w\end{aligned}\qquad(4.2)$$

これらの式は図4.1のパターンのように描くことができるが，これによって入れ子になった再帰的構造が明らかになり，また，これらのパターンは(4.2)の式の記憶領域上への配置あるいは写像を定めている．

再帰的情報構造の次の例として家系図(pedigree)について述べよう：**家系図**はある人(の名前)とその両親の家系図によって定義されるとしよう．この定義によると必ず無限構造になるが，実際の家系図は何代か前の先祖の情報が失われているので有限になる．この事実は再び可変構造を用いることにより(4.3)のように表現することができる．

```
type ped = record
               if known then
                  (name: alfa;
                   father, mother: ped)
           end                                    (4.3)
```

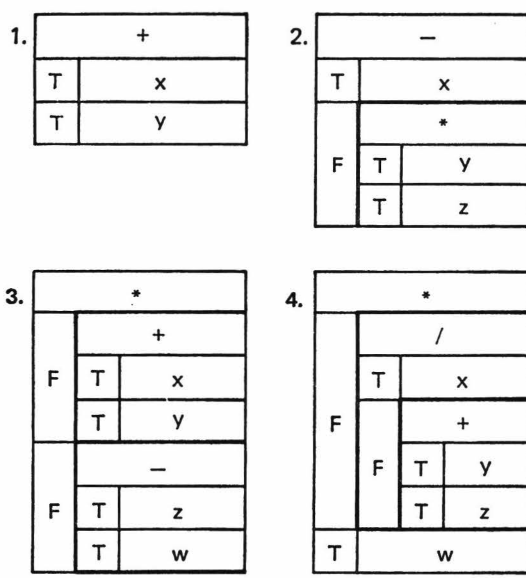


図4.1　再帰的レコード構造に対する記憶パターン

(型 *ped* のどの変数も少なくとも1つの成分，すなわち ***known*** と呼ばれるタグ・フィールドをもっていることに注意せよ．もしその値が ***true*** ならば，さらに3つのフィールドがあるが，そうでなければ他にはフィールドはない)．(再帰的) レコード構成子によって表わされる特別な値

$$x = (T, Ted, (T, Fred, (T, Adam, (F), (F)), (F)),\\ (T, Mary, (F), (T, Eva, (F), (F))))$$

は図**4.2**のように描かれるが，これは可能な記憶パターンを示唆している（た

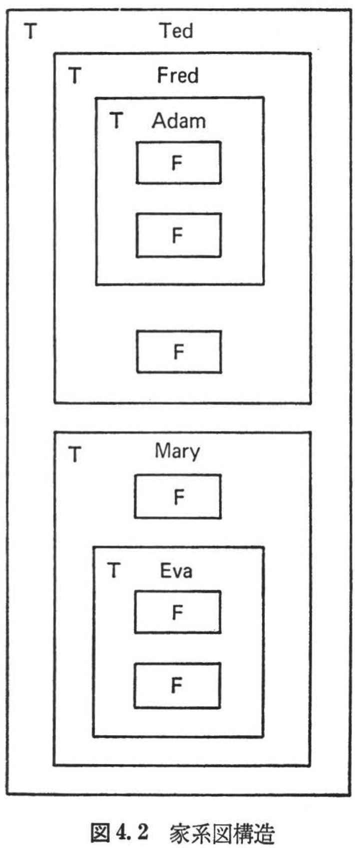


図4.2　家系図構造

だ1つのレコード型しか関係しないので，各構成子の前の型識別名 *ped* は省略した)．

可変レコード構造の重要な役割は明らかである：それは再帰的データ構造が有限になり得る唯一の方法であり，それゆえにすべての再帰的定義に必ず付随する．プログラムとデータに関する構造化の概念の類似性は，この場合には特に強い．再帰的手続きの実行が停止するためには，条件文がその手続きに必ず含まれていなければならない．実行の停止は明らかに有限基数に対応する．

4.2　ポインタまたは参照

再帰的構造が基本的構造（配列，レコード，集合）と明確に異なる特徴的性質は，その大きさが変化する能力である．それゆえ再帰的に定義される構造に対しては，一定の大きさの記憶領域を割り付けることが不可能であり，その結果，そのような変数の成分に対して，コンパイラは特定の番地を対応させることはできない．この問題を解決するために最も普通に使われる手法は記憶の**動的割付け**である．これは各成分への記憶割付けを，コンパイル時点ではなく，プログラムの実行中にそれが生じた時点で行なうものである．この場合，コンパイラは動的に割り付けられる成分の**番地**のために一定量の記憶領域を割り付けるが，成分そのもののための割付けは行なわない．たとえば図4.2に示される家系図は，各人につき1個の別々の――多分連続していない――レコードによって表現されるであろう．このとき各人は *father* および *mother* フィールド中に格納された番地によって結合される．図的にはこの状況は矢印または**ポインタ**(pointer)を使って表わすのが最もわかりやすい(**図4.3**参照)．

再帰的構造を実現するためのポインタの使用が，単なる手法に過ぎないことは強調しておかなければならない．プログラマはその存在を知っている必要はない．新しい成分が参照される最初の時に記憶領域が自動的に割り付けられればよい．しかしながら，もし**参照**(reference)あるいはポインタを使用するという手法が陽に表現される場合には，純粋に再帰的なデータの定義によって定められるものよりも，より一般的なデータ構造が構成され得る．特にこのときには，"無限の"または環状的構造を定義したり，あるいはある構造を**共有**したりすることができる．したがって，高度なプログラム言語ではデータ自身の他に，データに対する参照も陽に操作できるようになっているのが普通である．このためには，データとデータに対する参照を記法的に明確に区別することが必要であり，その結果，その値が他のデータへのポインタ(参照)であるようなデータ型が導入されなければならない．この目的のためにわれわれが用いる記法は，

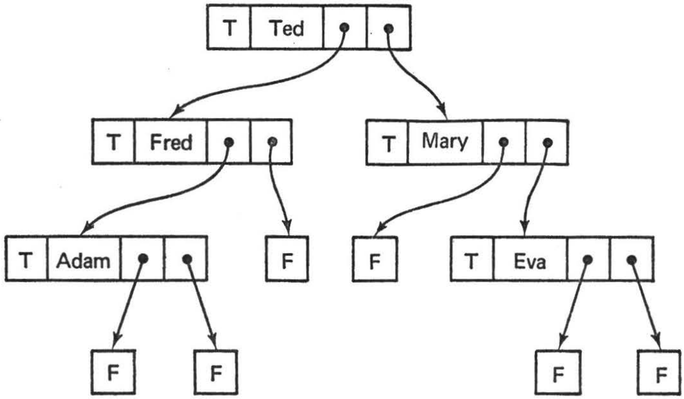


図 4.3 ポインタによって結合された構造

次のようなものである：

$$\textbf{type}\ T_p = \uparrow T \tag{4.4}$$

型宣言(4.4)は，型 T_p の値は型 T のデータへのポインタであることを表わしている．したがって(4.4)の矢印は“...へのポインタ”を表わしている．指されている要素の型が T_p の宣言から明白であることは基本的に重要である．われわれは T_p は T に**束縛**されているという．この束縛は高級言語におけるポインタを，アセンブラ語における番地から区別するものであり，記法上の冗長性を利用して，プログラム言語における安全性を増加させるための最も重要な機構の１つである．

ポインタ型の値は，データ項目が動的に割り付けられるたびごとに生成される．われわれは，そのような動的割付けは常に陽に述べなければならない，という規約に従うことにしよう．これは，ある項目が参照される最初のときに自動的に割り付けられる(と仮定する)という状況とは対称的である．この目的のために基本手続き *new* を導入する．型 T_p のポインタ変数 p が与えられたとき，文

$$new(p) \tag{4.5}$$

は，型 T の変数を実際に割り付け，この新しい変数を参照する型 T_p のポインタを生成し，そしてこのポインタを変数 p に代入する (**図 4.4** 参照)．したがって，このポインタ値自身は p として(すなわち，変数 p の値として) 参照

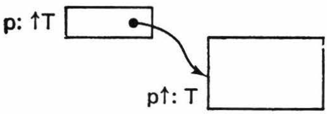


図4.4 変数 $p\uparrow$ の動的割付け

することができる．これに対し，p によって参照される変数を $p\uparrow$ で表わす．これは動的に割り付けられた型 T の変数である．

どのような再帰的型においても，有限の基数を保証するためには可変成分が本質的であることを既に述べた．家系図の例は最も頻繁に現われるパターンであるが，この場合には，タグ・フィールドは2値（論理値）であり，その値が *false* の時にはそれ以外の成分は存在しない．これは一般に(4.6)の宣言形式によって表現される．

$$\textbf{type } T = \textbf{record if } p \textbf{ then } S(T) \textbf{ end} \tag{4.6}$$

$S(T)$ はフィールド定義の列であり，その中には型 T の1つ以上のフィールドが含まれており，これによって再帰性が導入されている．(4.6) の形をした型をもつどの構造も図4.3で示されたものと同様な木(またはリスト)構造になる．それに特有な性質は，タグ・フィールド以外の情報を持たないデータ成分へのポインタを含んでいることである．ポインタを用いた実現方法では，タグ・フィールドの情報をポインタ自身の値に含めることにより，記憶領域を節約する簡単な方法が考えられる．よく使われる方法は，型 T_p の値域を**拡張**して，存在しない要素を指す値を1つ追加することである．我々はこの値を特別な記号 **nil** で表わし，この **nil** は宣言された全てのポインタ型に自動的に属していると考える．ポインタの値域に対するこの拡張は，構造の（再帰的な）宣言において，可変条件を陽に述べなくても有限構造を生成し得る理由を説明している．

(4.1)および(4.3)において宣言されたデータ型に対する——陽にポインタを用いた——新しい形式化が，(4.7)と(4.8)に与えられている．（原理的には形式(4.6)に対応している）後者の場合には，可変レコード成分が無くなっているが，これは $p\uparrow.known = false$ が $p = \textbf{nil}$ と表現されたからである．また，*ped* から *person* への型の名前がえは，陽なポインタの導入による観点の相違を反映させるためのものである．与えられた構造全体について最初に考え，次にその部分構造や成分について調べるという代わりに，新しい形式化ではまず第一にその成分に注意が向けられており，それらの（ポインタによって表現される）相互関係は，その静的な宣言からは明らかではない．

```
type expression  = record op: operator;
                       opd1, opd2: ↑term
                   end;
type term        = record                                    (4.7)
                       if t then (id: alfa)
                            else (sub: ↑expression)
                   end
```

```
type person      = record name: alfa;
                       father, mother: ↑person               (4.8)
                   end
```

図 4.2 と 4.3 で示された家系図を表わすデータ構造が，再び図 4.5 に与えられているが，この図では不明な人に対するポインタは **nil** によって表わされている．これによる記憶領域の節約は明らかである．

さて**図 4.5** において，Fred と Mary が兄弟である場合，すなわち共通の両親を持つ場合を考えよう．この状況は，2つのレコードのフィールド中の2つの **nil** 値を置き換えることにより，容易に表わすことができる．ポインタの概念を隠したり，あるいは異なった記憶操作手法を用いた実現方法では，Adam と Eva のレコードを各々2回表現することが必要になる．検索のためにデータにアクセスする場合には，父親(母親)が2回重複して表現されていても，あるいは1つのレコードによって表現されていても同じであるが，**選択的更新が許されている場合**には，その差は**本質的**である．ポインタを隠さず陽にデータ

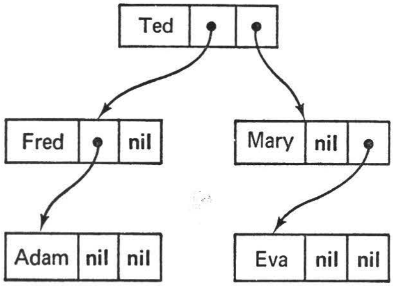


図 4.5　nil ポインタをもつ構造

項目として扱うことにより，プログラマは**記憶領域の共有**を明確に表現することができる．

さらに，ポインタを陽に表現することにより環状データ構造の定義と操作が可能になる．もちろん，この柔軟性の追加は計算能力を増加させるが，それだけではなく，プログラマーにより一層の注意力を要求する．なぜなら，環状データ構造の操作により，停止しない計算プロセスが容易に生じるからである．

能力と柔軟性は誤用による危険と背中合わせであるという現象は，プログラミングにおいて良く知られているが，特に **goto** 文の場合が想起される．実際，もしプログラム構造とデータ構造間の類似性を拡張するのであれば，純粋に再帰的なデータ構造は手続きに対応するレベルに置かれるべきであり，一方ポインタの導入は **goto** 文の使用に対応する．なぜなら，**goto** 文によっていかなる種類のプログラム・パターン（ループも含めて）の構成も可能であり，同様に，ポインタによって任意の種類のデータ構造(環状リストを含めて)の構成が可能だからである．表4.1にプログラムとデータ構造の対応が簡潔に並記されている．

表4.1　プログラムとデータ構造の対応

構成パターン	**プログラム文**	**データ型**
原子要素	代入文	スカラ型
数え上げ	複合文	レコード型
既知の回数の繰返し	for 文	配列型
選　択	条件文	可変レコード，合併(union)型
未知の回数の繰返し	while 又は repeat 文	列またはファイル型
再　帰	手続き文	再帰的データ型
一般の“グラフ”	goto 文	ポインタによって結合された構造

繰返しは再帰の特別の場合であり，形式(4.9)に従って定義される再帰的手続き P

```
procedure P;
begin
    if B then begin P0; P end
end
```

の呼出しは，繰返し文

$$\textbf{while } B \textbf{ do } P_0$$

に等価であり，置換え可能であることが3章で示されていた．ただし，P_0はPを含まない文である．

表4.1で略述された類似性から，再帰的データ構造と列データの間にも同様の関係が成立することがわかる．事実，形式

$$\begin{aligned}&\textbf{type } T = \textbf{record}\\&\qquad\qquad \textbf{if } B \textbf{ then } (t_0\textbf{: } T_0\textbf{; } t\textbf{: } T)\\&\qquad\quad \textbf{end}\end{aligned} \tag{4.10}$$

に従って定義される再帰的データ型は，列データ型

$$\textbf{file of } T_0$$

に等価であり置換え可能である．ただし，T_0はTを含まない型とする．このことは，プログラムとデータの再帰的定義において，もしその手続き名あるいは型名が，その定義中に1度だけその終わり（又は始め）に現われる時には（かつその時に限り），再帰は繰返しによって置換へ可能であることを示している．

本章のこれ以後の部分では，陽なポインタによって成分の結合されたデータ構造の生成と操作について述べる．特定の簡単なパターンをもった構造が特に強調されているが，その理由は，複雑な構造を取り扱うための手法は基本的構造を操作するための手法から導かれるからである．これらの簡単な構造とは，線形リストあるいは連結された列——これは最も簡単な場合である——および木である．データの構造化に関してわれわれがこれらの基本的構造を重要視したからといって，より複雑な構造が実際に起こらないということを意味するものではない．実際，次のような話が1922年7月のチューリッヒの新聞に載っていたが，これは木(家系図木)のように通常は規則的な構造の例として使われるようなものにも，不規則性が起こり得るという証拠である．この話はある不幸な男について次のように述べたものである．

> 私はある未亡人と結婚したが，彼女には成長した娘がいた．私の父は頻繁にわれわれのところに訪ねてきて，私の義理の娘と恋に落ち結婚した．したがって，父は私の養子となり，義理の娘は私の母となった．数ケ月後，私の妻は息子を生み，したがって彼は父の義弟であると同時に私の叔父となった．私の父の妻，すなわち私の義理の娘もま

た息子を生んだ．これにより，私には兄弟ができ，同時に孫もできた．私の妻は私の祖母である．なぜなら，彼女は私の母の母だからである．ゆえに，私は私の妻の夫であると同時に彼女の義理の孫である．言い換えれば，私は自分の祖父である．

4.3　線形リスト

4.3.1　基本操作

ひと組の要素を関連づけるあるいは結合する最も簡単な方法は，それらを1つのリストまたは列の形に整列させることである．なぜなら，この場合には直後の要素を参照するのにただ1つのリンクしか必要ではないからである．

ある型 T が (4.11) に示されるように定義されたと仮定しよう．この型のどの変数も3個の成分，すなわち識別キー，直後の要素 (successor) へのポインタおよび(4.11)では省略されているが，さらに何かしらの関連情報，から構成されている．

$$\begin{aligned}\textbf{type } T = \textbf{record } & \mathit{key}\colon \mathit{integer}; \\ & \mathit{next}\colon \uparrow T; \\ & \cdots\cdots \\ \textbf{end} & \end{aligned} \qquad (4.11)$$

型 T のリストの1例が**図4.6**に示されているが，この図ではリストの最初の成分へのポインタが変数 p に代入されている．図4.6のリストに対する最も簡単な操作は，たぶんその**先頭への要素の挿入**であろう．まず，型 T の1つの要素が割り付けられ，その参照値(ポインタ)が補助的ポインタ変数，たとえば q，に代入される．その後にポインタの単純な再代入を行なってこの操作は完了する．以上の動作は，(4.12)のようにプログラムされる．

$$new(q)\,;\ q\uparrow.next := p; p := q \qquad (4.12)$$

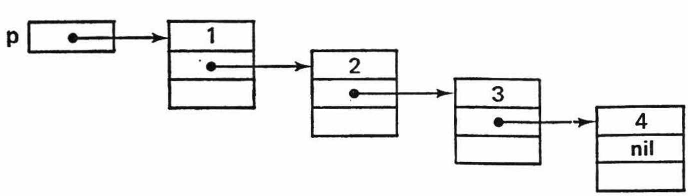


図4.6　リストの例

これらの3つの文の順序は本質的であることに注意せよ．

リストの先頭への要素の挿入操作から，ただちにリストの生成方法が示唆される：まず空リストから始めて，先頭に要素を次々と挿入してゆけばよい．このリスト生成の過程は(4.13)に示されている．ただし，結合されるべき要素の数は n である．

```
p := nil;  {start with empty list}
while n > 0 do
    begin new(q); q↑.next := p; p := q;             (4.13)
        q↑.key := n; n := n-1
    end
```

これはリストを形成するための最も簡単な方法である．しかしながら，生成されたリスト中での要素の順番は，“到着”の順番とは逆である．場合によってはこれは望ましくなく，新しい要素はリストの**終わり**に付けなければならない．リストの終わりは，それを走査することによって容易に決定できるが，リストの最後の要素を常に指している第2のポインタ，たとえば q，を使用することにより，走査のための作業を節約することができる．この方法の応用例が**プログラム4.4**に与えられているが，これは与えられたテキストに対する**相互参照**（cross-reference）を生成するプログラムである．この方法の不利な点は，最初の要素の挿入とそれ以後の要素の挿入とを別々に扱わなければならないことである．

他の方法では厄介であるが，ポインタを陽に使用することにより非常に簡単になる操作がいくつかある．初等的なリスト操作のなかでは，要素の挿入と削除（リストの選択的更新），およびリストの走査がそのようなものである．まず**リストへの挿入**から考えよう．

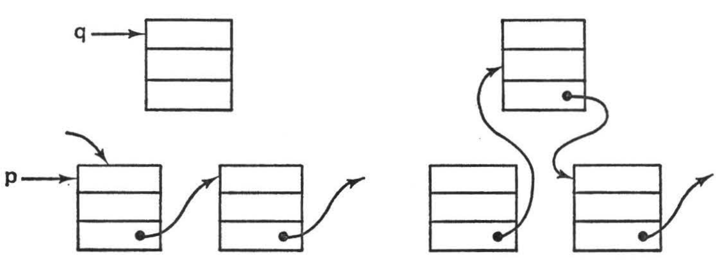


図4.7　$p\uparrow$ の直後への挿

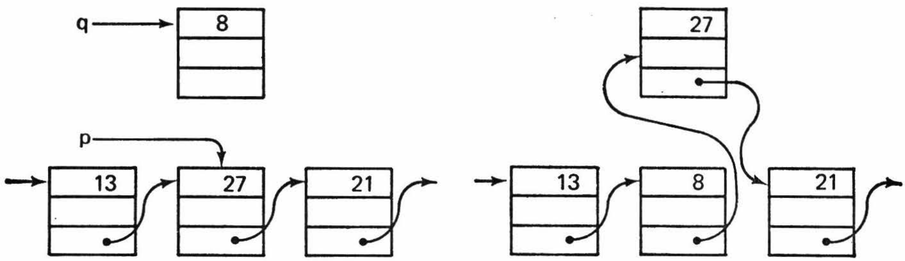


図 4.8　$p\uparrow$ の直前への挿入

あるリスト中のポインタ p で指される要素の**直後**に，ポインタ（変数）q で指されているある要素を挿入する問題を考えよう．必要となるポインタ代入は(4.14)で表わされ，その効果は**図 4.7** に図示されている．

$$q\uparrow.next := p\uparrow.next \; ; p\uparrow.next := q \tag{4.14}$$

もし，指示された要素 $p\uparrow$ の直後ではなく，**直前**への挿入が必要であるときには，1方向性リンク結合は問題を起こしそうに思える．なぜなら，ある要素からその直前の要素 (predecessor) へのどのような道も存在しないからである．しかしながら，簡単な“トリック”によってこの難問を解決することができる：それは(4.15)で与えられ**図 4.8** に図示されている．新しい要素のキーは $k=8$ とする．

$$\begin{aligned}&new(q)\,;q\uparrow := p\uparrow;\\&p\uparrow.key := k;\; p\uparrow.next := q\end{aligned} \tag{4.15}$$

明らかにこの“トリック”では新しい要素の挿入は $p\uparrow$ の直後であるが，新しい要素の値と $p\uparrow$ の値とが交換されている．

次に**リストからの削除**の過程を考えよう．$p\uparrow$ の直後の要素の削除は簡単である．(4.16)にはこの過程および削除された要素を他のリストの先頭（q で指されている）に再び挿入する過程とが一緒に示されている．r は型 $\uparrow T$ の補助変数である．

$$\begin{aligned}&r := p\uparrow.next;\; p\uparrow.next := r\uparrow.next;\\&r\uparrow.next := q;\; q := r\end{aligned} \tag{4.16}$$

図 4.9 は(4.16)の過程を図示したものであるが，3つのポインタが巡回的に交換されていることが示されている．

指示された要素自身（その直後の要素ではなく）の除去は，$p\uparrow$ の直前への挿入と同じ問題に遭遇するのでより困難である：すなわち，指示された要素の直

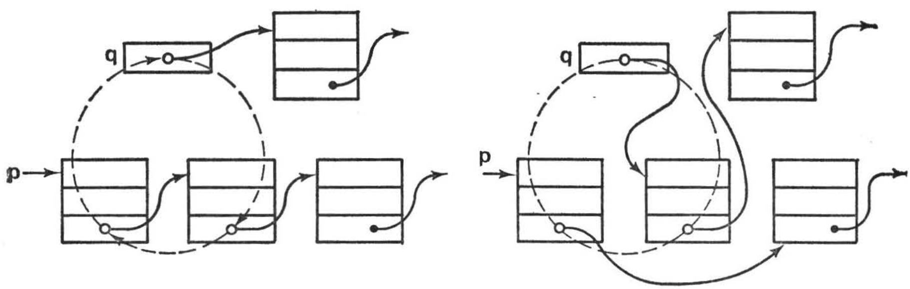


図 4.9 リストからの削除と再挿入

前のものへの後戻りは不可能だからである．しかし，直後の要素の値を前側に移した後でそれを除去する方法は，比較的明らかで簡単な解である．この方法は $p\uparrow$ が直後の要素を持つ場合，すなわちリストの最後の要素でない場合に適用可能である．

さて，最後の基本操作**リストの走査**について考えよう．その最初の要素が $p\uparrow$ であるようなリストのすべての要素に対して操作 P を施すとしよう．この作業は次のように表現される．

> **while** *list designated by p is not empty* **do**
> **begin** *perform operation P*;
>   *proceed to the successor*
> **end**

そしてこれは，文(4.17)のように詳細化される．

> **while** $p \neq$ **nil do**
>   **begin** $P(p\uparrow)$; $p := p\uparrow.next$ (4.17)
>   **end**

while 文とリンク構造の定義から，P はリスト中のすべての要素に適用されるが，それ以外のものには適用されないことがわかる．

このような操作のうちで非常に頻繁に実行されるものに，与えられたキー x をもつ要素に対する**リストの探索**がある．ファイル構造の場合と同様に探索は順次的である．探索はそのような要素が見つけられるか，またはリストの終わりに到達した時に終了する．再び，リストの先頭はポインタ p によって指されていると仮定しよう．

この単純な探索の定式化の第一段階として，次のものが得られる．

$$\textbf{while}\,(p \neq \textbf{nil}) \wedge (p\uparrow.key \neq x)\,\textbf{do}\;\; p := p\uparrow.next \qquad (4.18)$$

しかしながら，p=**nil** は $p\uparrow$ が存在しないことを意味することに注意しなければならない．したがって，停止条件の評価によって，存在していない変数（未定義値をもつ変数とは異なる）にアクセスし，プログラムの実行エラーを起こす可能性がある．これは，(4.19)のように **goto** 文によって繰返しの中断を陽に表現するか，または(4.20)のように目的のキーが見つけられたか否かを表わす論理変数を導入することにより避けることができる．

```
while p ≠ nil do
    if p↑.key = x then goto Found
                  else p := p↑.next                    (4.19)
```

goto 文を使うためには，どこかに行先のラベルが必要である．**goto** 文は **while** 文と両立し難いことに注意してほしい．なぜなら，while 節が誤解を招きやすくなるからである：すなわち，たとえ $p \neq$ **nil** であっても，(**goto** *Found* が実行された後では†) 制御される文 (**if** 文†) は実行されない††.

```
b := true;
while (p ≠ nil) ∧ b do
    if p↑.key = x then b := false
                  else p := p↑.next                    (4.20)
{(p=nil) ∨ ¬b}
```

4.3.2　整列リストおよびリストの再構成

アルゴリズム(4.20)は，配列やファイルを走査するための探索ルーチンを強く想い起こさせる．実際，ファイルは，直後の要素への連結手法が未指定であるかまたは陰に隠されている線形リストに他ならない．基本ファイル操作としては，(終端を除いては)新しい要素の挿入は許されないし，また（**すべて**の要素の除去以外には）要素の削除は許されない．したがって，実現方法に対して多様な選択の余地が残されており，連続記憶領域に要素を順配置することも可

† 訳者挿入

†† 訳注　**while** 文の意味を単純に“繰返し条件が成立している間実行する”とすると **goto** 文によって **while** 文から抜け出す場合には，この時点でも繰返し条件が成立している可能性があるので，この“単純”な原則に反することになる．

能である．ポインタを陽に用いた線形リストは**より多くの柔軟性**をもっており，したがってこの余分の柔軟性が必要なときにはいつでも使われるべきである．

実例を示すためにある問題を考えるが，この問題は別の解や手法を示すために本章を通して使用される．これは，あるテキストを読み，それに含まれる語とその出現頻度を数えるという問題であり，**用語索引** (concordance) の作成と呼ばれる．

1つの明らかな解は，テキスト中で見つけられた語からなるリストを構成することである．各語に対してこのリストを走査し，もしその語があればその頻度を増加させる：もしなければ，この語をリストに加える．この操作の中には明らかに挿入も含まれるが，われわれはこれを単に**探索**と呼ぶことにする．

リストの取扱いの本質的な部分に集中できるように，各語は調査中のテキストから既に抽出されており，整数に符号化されて入力ファイルの形で存在すると仮定する．

search (探索) と呼ばれる手続きの定式化が (4.20) からただちに得られる．変数 *root* は，新しい要素の挿入が (4.12) に従って行なわれるようなリストの先頭を指している．完全なプログラムが**プログラム 4.1** に与えられているが，これには構成された用語索引リストの作表ルーチンも含まれている．作表の過程は，リストの各要素に対してある動作が1度だけ行なわれる (4.17) の形式の1例になっている．

プログラム 4.1 の線形走査アルゴリズムは，配列とファイルに対する探索手

プログラム 4.1 リストへの単純挿入

```
program list (input,output);
{straight list insertion}
    type ref = ↑word;
        word = record key: integer;
                      count: integer;
                      next: ref
                end ;
    var k: integer; root: ref;

    procedure search (x: integer; var root: ref);
        var w: ref; b: boolean;
    begin w := root; b := true;
        while (w≠nil) ∧ b do
            if w↑.key = x then b := false else w := w↑.next;
```

```
        if b then
        begin {new entry} w := root; new (root);
            with root↑ do
            begin key := x; count := 1; next := w
            end
        end else
        w↑.count := w↑.count + 1
    end {search} ;
    procedure printlist (w: ref);
    begin while w ≠ nil do
            begin writeln (w↑.key, w↑.count);
                    w := w↑.next
            end
    end {printlist} ;
begin root := nil; read(k);
    while k ≠ 0 do
        begin search (k, root); read(k)
        end ;
    printlist(root)
end
```

続き，特にループ停止条件を簡単化するために用いられた簡単な手法すなわち**番人**の使用を想い起こさせる．番人はリスト探索の場合にも用いることができる：それはリストの最後に置かれたダミーの要素によって表現される．番人を考慮した新しい手続きが(4.21)に与えられており，プログラム4.1中の探索手続きと置き換えられる．ただし，大域変数 *sentinel* (番人)が新たに加えられており，かつ *root* の初期化は次の文で置き換えられるとする．

new (*sentinel*) ; *root* : = *sentinel*;

これは番人として使われる要素を生成する．

```
procedure search(x: integer; var root: ref);
    var w: ref;
begin w := root; sentinel↑.key := x;
    while w↑.key ≠ x do w := w↑.next;
    if w ≠ sentinel then w↑.count := w↑.count + 1 else          (4.21)
        begin {new entry} w := root; new(root);
```

```
            with root↑ do
            begin key := x; count := 1; next := w
            end
        end
end {search}
```

明らかに，この例では連結リストの能力と柔軟性が正しく使われていないし，リスト全体を線形走査するのが許されるのは，要素の数が限られている場合だけである．しかしながら，1つの簡単な改良がすぐ手近なところにある：すなわち，**整列リストの探索**である．もし，リストが整列している（たとえばキーの大きさの順に）ならば，新しいキーより大きい最初のキーに出合った時点で探索を停止することができる．リストの整列性は，新しい要素をリストの先頭ではなく適当な場所に挿入することにより保つことができる．実際，整列性は"ただ"で得られる．これは，連結リストへの挿入の容易さによるものであり，その柔軟性を最大に利用した結果である．これは配列やファイル構造では不可能である．（しかしながら，たとえば整列リストでも，配列の2分探索に対応するものはないことに注意せよ）．

整列リスト探索は(4.15)で述べた状況の典形的な例である．すなわち，新しい要素を与えられた項目の直前に，すなわちそのキーが新しいキーより大きい最初のものの直前に挿入しなければならない．しかし，ここで示す技法は，(4.15)で用いられたものとは異なる．値をコピーする代わりに，2つのポインタをリストの走査で用いる：$w2$ は $w1$ より1ステップ遅れており，したがって，$w1$ が大きすぎるキーを見つけた時には，$w2$ は正しい挿入場所を示している．一般的な挿入ステップが**図 4.10** に示されている．

先に進む前に，2つの状況について考えておかなければならない．

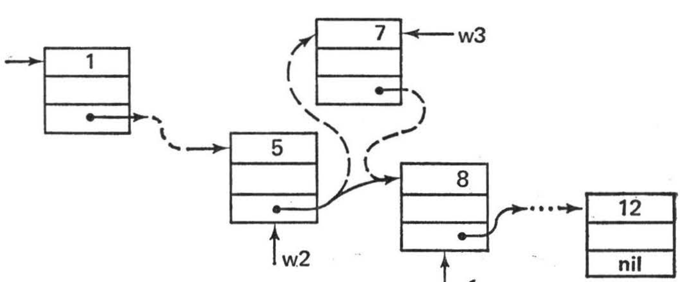


図 4.10 整列リストへの挿入

1. リストが空である場合を除いては，新しい要素へのポインタ ($w3$) 値は $w2\uparrow.next$ に代入されなければならない．単純性および有効性の理由から，条件文を使ってこの区別をすることは好ましくない．これを避ける唯一の方法は，リストの先頭にダミーな要素を導入することである．
2. 2つのポインタが1ステップだけ離れながらリストを走査するためには，(ダミーの他に)少なくとも1個の要素がリストに含まれていなければならない．これは，最初の要素の挿入は残りのものとは別に扱われることを意味する．

これらの指針とヒントから生まれた1つの提案が(4. 23)に与えられている．これは補助手続き *insert* を使用するが，この手続きは *search* に対して局所的であると宣言されているとする．*insert* は (4. 22) に示されており，新しい要素 w の生成と初期化を行なう．

```
procedure insert(w: ref);
   var w3: ref;
begin new(w3);
   with w3↑ do
      begin key := x; count := 1; next := w                    (4. 22)
      end ;
   w2↑.next := w3
end {insert}
```

プログラム 4. 1 中の初期化文 "*root* :=**nil**"は次のもので置き換える．

$$new(root)\ ;\ root\uparrow.next := \textbf{nil}$$

図 4. 10 を参照して，走査を次の要素に進める条件を決定すると，これは2つの因子から成っている．すなわち，

$$(w1\uparrow.key < x) \land (w1\uparrow.next \neq \textbf{nil})$$

この結果，探索手続きは(4. 23)に示すようになる．

```
procedure search(x: integer; var root: ref);
   var w1,w2: ref;
begin w2 := root; w1 := w2↑.next;
   if w1 = nil then insert (nil) else

                                                               (4. 23)
```

```
  begin
     while (w1↑.key < x) ∧ (w1↑.next ≠ nil) do
        begin w2 := w1; w1 := w2↑.next
        end ;
     if w1↑.key = x then w1↑.count := w1↑.count + 1 else
        insert(w1)
  end
end {search} ;
```

残念なことにこの提案には欠陥がある．注意したにもかかわらず，“虫”が入り込んでしまった！ 先に進む前に是非，読者自ら誤りを発見してほしい．捜査官の任務を放棄することを選んだ読者には，(4.23)は最初に挿入される要素をいつもリストの末尾に押し込むと言えば十分であろう．走査が2番目の因子によって停止する場合には，新しい要素は *w*1↑ の**前**ではなく後に挿入されなければならないことを考慮することにより，この欠陥は訂正される．ゆえに，文“*insert*(*w*1)”を次のもので置き換える．

```
begin if w1↑.next = nil then
         begin w2 := w1; w1 := nil
         end;                                        (4.24)
      insert(w1)
end
```

意地の悪いことに，信じやすい読者はまたもだまされたことになる．なぜなら，(4.24)もまだ正しくないからである．今度の誤りを知るために，新しいキーが最後のキーとその直前のキーの間にあると仮定しよう．この場合には走査がリストの終端に達したときには，継続条件の2つの因子がともに *false* になり，したがって最後の要素の後ろに挿入が行なわれてしまう．もし，同じキーが後で再び発生した場合にはこれは正しく挿入され，ゆえに表には2回現われるであろう．これを直すためには，(4.24)中の条件

$$w1\uparrow.next = \textbf{nil}$$

を次のもので置き換えればよい．

$$w1\uparrow.key < x$$

探索の高速化のために，再び番人を用いることにより，**while** 文の継続条件の簡単化を行なうことができる．このためには，末尾の番人と同様に，**ダミーのヘッダー**が初めから存在していることが必要である．ゆえに，リストは次の文

によって初期化されなければならない.

$$new(root)\ ;\ new(sentinel)\ ;\ root\uparrow.next := sentinel;$$

このとき，探索手続きは(4. 25)に見られるように著しく簡単になる.

```
procedure search(x: integer; var root: ref);
    var w1,w2,w3: ref;
begin w2 := root; w1 := w2↑.next; sentinel↑.key := x;
    while w1↑.key < x do
        begin w2 := w1; w1 := w2↑.next
        end ;
    if (w1↑.key = x) ∧ (w1 ≠ sentinel) then
        w1↑.count := w1↑.count + 1 else
    begin new(w3); {insert w3 between w1 and w2}                (4. 25)
        with w3↑ do
            begin key := x; count := 1; next := w1
            end ;
        w2↑.next := w3
    end
end {search}
```

さて，整列リストの探索から期待される利益について考えよう．必要となる余分の手間が少ないことを考えると，圧倒的な改善を期待すべきではない.

テキスト中のすべての語が同じ頻度で起こると仮定する．この場合，すべての語がリストされた後では，整列性から得られる利益はない．なぜなら，もしアクセス・ステップの総数のみに意味があり，かつすべての語が同じ出現頻度をもっているのであれば，語の位置は関係ないからである．しかしながら，新しい語が挿入されるときには利益がある．リスト全体を走査する代わりに，平均的にはリストの半分だけを走査すればよいからである．したがって，各語の出現頻度に比べて語の種類が多い場合の用語索引を作る場合にのみ，整列リスト法は利益がある．したがって，これまで考えてきた例題は，実際的応用というより，主にプログラミングの練習問題として適当なものである.

連結リスト中にデータを配置する方法は，要素数が比較的少なく（たとえば <100)，可変であり，かつアクセスの頻度に関する情報が与えられていないときに推奨される．ひとつの典形的な例は，プログラム言語のコンパイラの記号表である．新しい識別名は宣言によってこのリストに加えられ，その有効性の範囲から出るとき削除される．比較的短いプログラムに対する応用としては,

単純な連結リストの利用が適当である．この場合でも，以下に述べる非常に簡単な技法を用いることにより，アクセス法についてかなりの改善がなされるが，これは連結リストの柔軟性を示す格好な例である．

プログラムの特徴的性質は，同じ識別名がかたまって現われやすいことである．すなわちある語が出現すると，その同じ語は引き続いて出現しやすい．このことから，探索によって見つけられた語をリストの先頭に移動して，リストを再組織化することが考えられるが，これにより，次にその語が探索される時の探索パスの長さを最小にすることができる．このアクセス法は，**再順序づけを伴なうリスト探索**あるいは――いくらかもったいぶって――**自己組織化リスト探索**と呼ばれる．次に，プログラム 4.1 中に置き換え可能な手続きの形でこのアルゴリズムを表現するが，このとき，これまでの経験から番人を初めから導入する．実際，番人は探索を高速化するだけではなく，この場合にはプログラムを簡単にする．したがって，リストは初めから空ではなく，既に番人のための要素を含んでいる．初期化文は次のようである．

$$new(sentinel)\,;\ root := sentinel\,;$$

新しいアルゴリズムとリストの単純な探索との主な相違点は，ある要素が見つけられたときの再順序化動作であることに注意しよう．この要素は古い位置から削除されて先頭に再挿入される．この削除のために 2 つの追跡し合うポインタが再び必要になるが，*w*1 は見つけられた要素を，*w*2 はその直前の要素を指す．その結果，最初の要素に対しては（すなわち空リストに対しては）特別な取扱いが要求される．再連結の過程を理解するために，**図 4.11** を参照しよう．これは，$w1\uparrow$ が求める要素であるときの 2 つのポインタを示している．正しい再順序づけ後の状態が**図 4.12** に表わされており，また新しい探索手続き

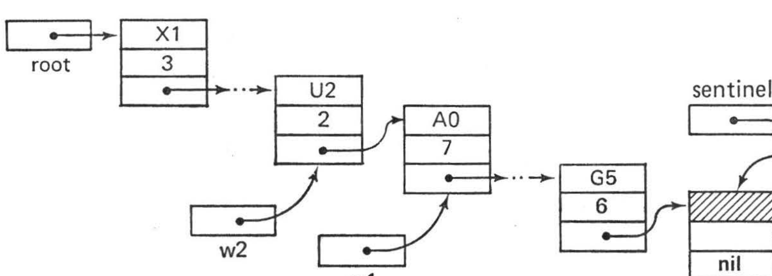


図 4.11　再順序づけ前のリスト

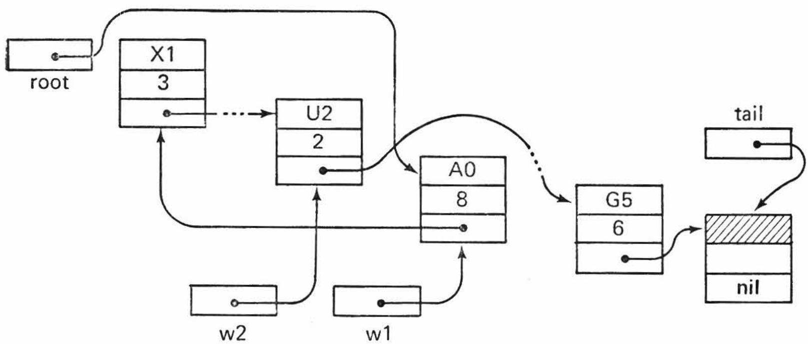


図 4. 12　再順序づけ後のリスト

```
procedure search(x: integer; var root: ref);
   var w1,w2: ref;
begin w1 := root; sentinel↑.key := x;
   if w1 = sentinel then
   begin {first element} new(root);
      with root↑ do
         begin key := x; count := 1; next := sentinel
         end
   end else
   if w1↑.key = x then w1↑.count := w1↑.count + 1 else
   begin {search}
      repeat w2 := w1; w1 := w2↑.next
      until w1↑.key = x;
      if w1 = sentinel then                                  (4. 26)
      begin {insert}
         w2 := root; new(root);
         with root↑ do
            begin key := x; count := 1; next := w2
            end
      end else
      begin {found, now reorder}
         w1↑.count := w1↑.count + 1;
         w2↑.next := w1↑.next; w1↑.next := root; root := w1
      end
   end
end {search}
```

表 4.2 リスト探索法の比較

	テスト 1	テスト 2
異なるキーの数	53	582
キーの出現回数	315	14341
線形リスト探索による時間	6207	3200622
再順序づけを伴なう探索による時間	4529	681584
改善度	1.37	4.70

の全体が(4.26)に与えられている.

この探索法による改善は，入力データ中の（同じ語の）かたまりぐあいに強く依存する．かたまりぐあいが同じ場合には，リストが大きい程改善は顕著である．どの程度の改善が期待できるかを知るために，上記の用語索引問題を短いテキストと比較的長いテキストに適用し，線形リスト法(4.21)とリスト再組織化法(4.26)の比較を行ない，経験的な測定値を得た．測定データは**表 4.2** に要約されている．残念なことに，最も大きな改善が得られるのは，どのみち別のデータ構成法が必要な場合である．4.4 節で再びこの例題に戻る.

4.3.3 応用：トポロジカル・ソート

柔軟で動的なデータ構造の適切な使用例の1つとして，**トポロジカル・ソート**の問題がある．これは，**半順序**が定義されている集合，すなわち項目のある対に対しては順序が定義されているが，必ずしもすべての対には順序が定義されていない集合をソートする問題である．これはよくある問題である．次のものは半順序集合の例である.

1. 辞書では，語は他の語を使って定義される．語 w が語 v を使って定義されるとき，$v \prec w$ と書く．辞書中の語をトポロジカル・ソートするとは，前向きの参照がないような順序に語を並べることである.
2. 作業(たとえば工学プロジェクト)は部分作業に分解される．ある部分作業を完了するためには，他の部分作業を先に実行しておかなければならないのが普通である．部分作業 v が部分作業 w より先に行なわれなければならないとき $v \prec w$ と書く．この場合，部分作業をトポロジカル・ソートするとは，各部分作業を開始するとき，それに必要な部分作業がすべて完了しているようにそれらを配列することを意味する.
3. 大学のカリキュラムでは，ある科目は他の科目より先にとらなければならないが，これはそこでの講義内容を他の科目で前提としているからであ

る．科目 v が科目 w の前提であるとき，$v \prec w$ と書く．このときトポロジカル・ソートにより，どの科目もそれより後の科目を前提とすることがないように科目の配置を行なうことができる．

4. プログラムでは，ある手続きは他の手続きの呼出しを含むことができる．手続き v が手続き w によって呼ばれるとき $v \prec w$ と書く．トポロジカル・ソートにより，前向きの参照が生じないように手続き宣言を配置することができる．

一般に，集合 S の半順序は，S の要素間のひとつの関係である．この関係は記号 $\prec$ [“先行する”(precede) と読む†] によって表わされ，S の任意の要素 x, y, z に対して，次の3つの性質(公理)を満たす．

(1)　$x \prec y$ かつ $y \prec z$ ならば $x \prec z$　　(推移律)

(2)　$x \prec y$ ならば $y \prec x$ ではない　　(反対称律)

(3)　$x \prec x$ ではない　　(反反射律)

明白な理由により，アルゴリズムによってトポロジカル・ソートされる集合 S は有限であると仮定しよう．したがって，その節点が S の要素を表わし，有向枝が順序関係を表わすようなグラフまたは図を描くことによって，半順序を表現することができる††．1例が **図4.13** に示されている．

トポロジカル・ソートの問題は，半順序を線形順序の中に埋め込むことである．グラフ的には，これは**図4.14** のように，すべての矢印が右側を向くようにグラフの節点を一列に並べることである．半順序の性質(1)，(2)はグラフがループを含まないことを保証する．これは正に，線形順序へのそのような埋め込みが可能であるための必要条件である．

どのようにすれば，そのような線形順序のひとつを見つけることができるであろうか．答は全く簡単である．まず初めに，他のいかなる項目にも先行されない任意の項目を選ぶ（このような項目は少なくとも1個は存在する．なぜなら，もしそうでないとループが存在することになる)．これを結果のリストの先頭に置き，S から除去する．残った集合はまた半順序集合であるから，S が空になるまで同じアルゴリズムを繰り返し適用すればよい．

このアルゴリズムをもっと厳密に記述するためには，S とその順序のデータ

---

† 訳注　“小さい”と読まれることも多い．

†† 訳注　この表現は少しあいまいである，図4.13，4.14では，順序関係は頂点間を結ぶ有向パスによって表現されている．したがって，原則的には，推移律によって生成されるような関係はこのようなグラフの枝として記述することは不必要である．ただし，図4.13では，冗長な枝が3本存在している (1→3, 4→8, 7→5)．

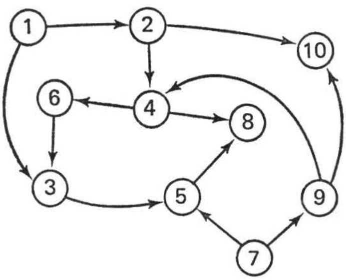


図 4.13 半順序集合

構造と表現を決定しなければならない．この表現の決定は，実行される操作，特に**先行要素** (predecessor) を持たない要素の選択操作によって決められる．ゆえに，すべての項目は3つの値によって表現されなければならない：これらは，同定キー (key)，**後続要素** (successor) の集合，および先行要素の個数 (count) である．S 中の要素の数 n は，前もっては与えられないので，この集合は連結リストによって構成するのが便利である．したがって，各項目の記述には，リスト中の次の項目へのポインタが追加されなければならない．キーは整数であると仮定する（しかし，1 から n までの連続した整数とは限らない）．同時に，各項目の後続要素の集合も連結リストで表現するのが便利である．後続要素リストの要素は，識別子およびこのリスト中の次の項目へのポインタによって記述される．主リストの記述子（ここには S の各項目がただ1回だけ現われる）を *leader* と呼び，後続要素リストの記述子を *trailer* と呼ぶことにすれば，データに関して次の宣言 (4.28) が得られる：

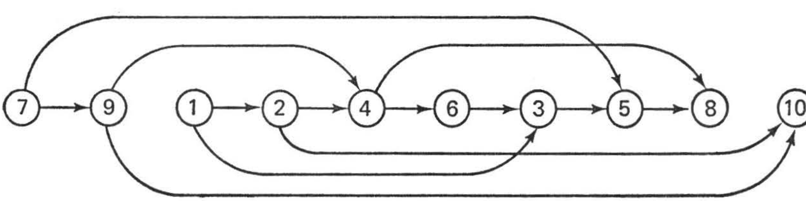


図 4.14 図 4.13 の半順序集合の線形配置

$$
\begin{aligned}
&\textbf{type } lref = \uparrow leader;\\
&\quad tref = \uparrow trailer;\\
&\quad leader = \textbf{record } key,\ count\colon integer;\\
&\qquad\qquad trail\colon tref;\\
&\qquad\qquad next\colon lref\\
&\qquad\quad \textbf{end};\\
&\quad trailer = \textbf{record } id\colon lref;\\
&\qquad\qquad next\colon tref\\
&\qquad\quad \textbf{end}
\end{aligned}
\tag{4.28}
$$

集合 S とその順序は，はじめはキーの対からなる列によって入力ファイル中に表現されていると仮定する．図4.13の例に対する入力データが (4.29) に示されているが，見やすさのために記号≺がつけられている．

$$
\begin{array}{ccccccc}
1 \prec 2 & 2 \prec 4 & 4 \prec 6 & 2 \prec 10 & 4 \prec 8 & 6 \prec 3 & 1 \prec 3\\
3 \prec 5 & 5 \prec 8 & 7 \prec 5 & 7 \prec 9 & 9 \prec 4 & 9 \prec 10 &
\end{array}
\tag{4.29}
$$

トポロジカル・ソート・プログラムの最初の部分では，入力ファイルを読み，このデータをリスト構造に変換しなければならない．これは，キー x と y の対 $(x \prec y)$ を次々に読むことにより行なわれる．*leader* からなる連結リストにおけるそれらの表現へのポインタを p と q で表わすことにする．これらのレコードは，リスト探索によって見つけられるが，もしまだ存在していなければリストに挿入される．この作業は，L と呼ばれる関数手続きによって実行される．次に，y の識別子をもった新しいエントリを x の *trailer* のリストに加える．そして，y の先行要素の個数を1増やす．このアルゴリズムは *input phase*（入力フェーズ）(4.30) と呼ばれる．**図4.15**は，(4.30) によって入力データ (4.29) を処理することにより生成されたデータ構造を示している．

```
{input phase} read(x);
new(head); tail := head; z := 0;
while x ≠ 0 do
begin read(y); p := L(x); q := L(y);                    (4.30)
   new(t); t↑.id := q; t↑.next := p↑.trail;
   p↑.trail := t; q↑.count := q↑.count + 1;
   read(x)
end
```

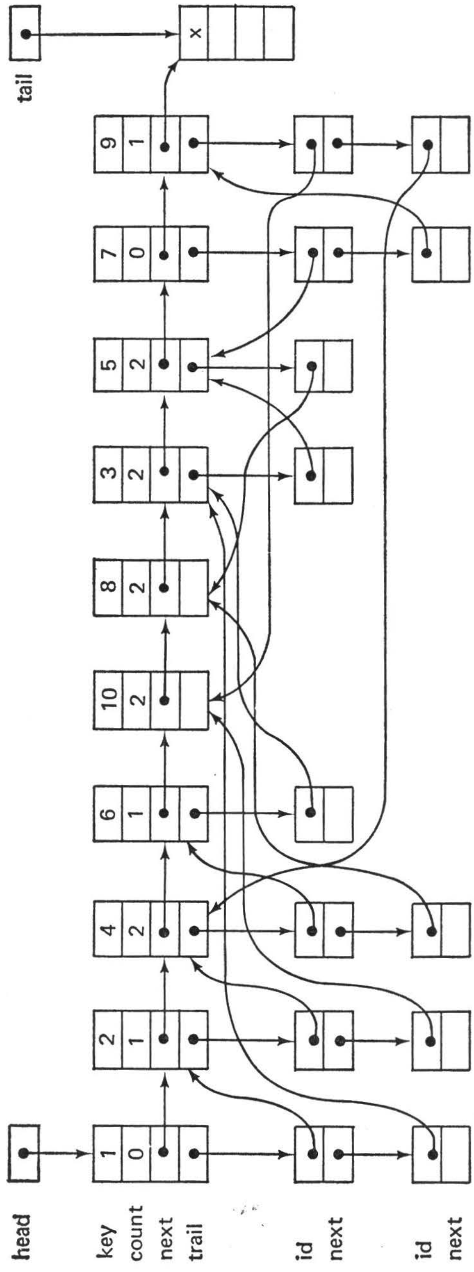


図 4.15 *topsort* プログラムにより生成されたリスト構造

この部分プログラムでは関数 $L(w)$ を参照しているが，これはキーが w であるリスト成分へのポインタをその値として返す(**プログラム 4.2** も参照せよ)．入力キー対の列の終わりには 0 が加えられていると仮定する．

図 4.15 のデータ構造がこの *input phase* で構成されてしまえば，次にはトポロジカル・ソートの実際の処理にとりかかることができる．しかしながら，そこでの作業は先行要素の個数が 0 である要素を繰り返し選択することであるので，そのような要素をあらかじめすべて集め，連結しておくことが賢明であると思われる．*leader* の初めのチェインは後には必要でなくなるので，*next* と呼ばれるこの同じフィールドを，先行要素が 0 である *leader* を連結するために再利用することができる．あるチェインを別のチェインで置き換えるというこのような操作は，リスト処理では頻繁に用いられる．これは (4.31) に詳細に述べられており，便宜上**逆方向**に新しいチェインを作り上げる．
図 4.15 については，*leader* の *next* チェインは**図 4.16** のそれによって置き換えられる．ただし描かれていないポインタは不変である．

```
{search for leaders with 0 predecessors}
  p := head; head := nil;
  while p ≠ tail do
     begin q := p; p := q↑.next;
        if q↑.count = 0 then                          (4.31)
        begin {insert q↑ in new chain}
           q↑.next := head; head := q
        end
     end
```

以上によって半順序集合 S に都合のよい表現の準備がすべて完了したので，いよいよトポロジカル・ソートの実際の作業，すなわち出力列を生成する作業に進むことができる．最初の概略版は次のように記述できる．

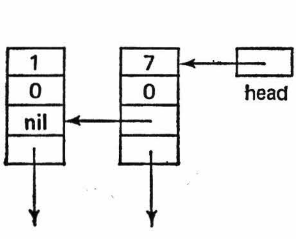


図 4.16　先行要素数が 0 の *leader* のリスト

```
q := head;
while q ≠ nil do
begin  {output this element, then delete it}
    writeln(q↑.key); z := z-1;
    t := q↑.trail; q := q↑.next;                        (4.32)
    "decrement the predecessor count of all its successors
     on trailer list t; if any count becomes 0, insert this
     element in the leader list q"
end
```

(4.32)中でさらに詳細化すべき文"decrement the predecessor..."("*trailer* リスト上にあるそれの後続要素すべての先行要素数を減らす：もし0になるものがあれば，この要素を *leader* リスト *q* に挿入する")には，リストの走査がもう1回必要である[形式(4.17)参照]．各ステップで，補助変数 *p* は先行要素数の減少と判定を行なうべき *leader* 要素を指している．

```
while t ≠ nil do
begin p := t↑.id; p↑.count := p↑.count-1;
      if p↑.count = 0 then
      begin  {insert p↑ in leader list}
           p↑.next := q; q := p                          (4.33)
      end;
      t := t↑.next
end
```

これでトポロジカル・ソートのプログラムが完成した．カウンタが *input phase* で生成された *leader* の数を数えするために導入されていたことに注意してほしい．この値は，*output phase*（出力フェーズ）において *leader* 要素が出力されるごとに減少され，プログラムの終わりでは0に戻らなければならない．これが0にならないということは，先行要素を持たない要素が存在しなくなったときに，この構造中にまだ要素が残っていることを示している．このような場合が生じるのは，明らかに *S* が半順序集合ではないときである．

以上のようにプログラムされた *output phase* は，脈動するリスト，すなわち予測できない順序で要素が挿入されたり削除されたりするリスト，を管理するプロセスの1例である．それゆえに，これは陽に連結されたリストのもつ柔軟性を十分に利用しているプロセスの例でもある．

プログラム 4.2　トポロジカル・ソート

```
program topsort(input,output);
type lref = ↑leader;
    tref = ↑trailer;
    leader = record key: integer;
                    count: integer;
                    trail:  tref;
                    next:  lref;
             end ;
    trailer = record id: lref;
                    next: tref
             end ;
var head, tail, p,q: lref;
    t: tref; z: integer;
    x,y: integer;
function L(w: integer): lref;
    {reference to leader with key w}
    var h: lref;
begin h := head; tail↑.key := w;  {sentinel}
    while h↑.key ≠ w do h := h↑.next;
    if h = tail then
       begin {no element with key w in the list}
          new(tail); z := z+1;
          h↑.count := 0; h↑.trail := nil; h↑.next := tail
       end ;
    L := h
end {L} ;
begin {initialize list of leaders with a dummy}
    new(head); tail := head; z := 0;
{input phase}  read(x);
    while x ≠ 0 do
    begin read(y); writeln(x,y);
       p := L(x); q := L(y);
       new (t); t↑.id := q; t↑.next := p↑.trail;
       p↑.trail := t; q↑.count := q↑.count + 1;
       read(x)
    end ;
{search for leaders with count = 0}
    p := head; head := nil;
```

```
    while p ≠ tail do
    begin q := p; p := p↑.next;
       if q↑.count = 0 then
          begin q↑.next := head; head := q
          end
    end ;
{output phase}  q := head;
    while q ≠ nil do
    begin writeln(q↑.key); z := z-1;
       t := q↑.trail; q := q↑.next;
       while t ≠ nil do
       begin p := t↑.id; p↑.count := p↑.count - 1;
          if p↑.count = 0 then
          begin {insert p↑ in q-list}
             p↑.next := q; q := p
          end ;
          t := t↑.next
       end
    end ;
    if z ≠ 0 then writeln ('THIS SET IS NOT PARTIALLY ORDERED')
end
```

4.4 木 構 造

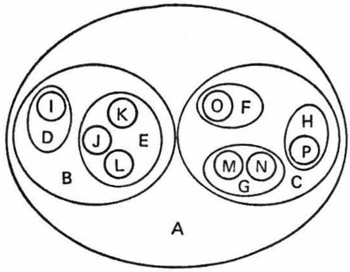
4.4.1 基本概念と定義

われわれはこれまでに，列とリストは次のように定義するのが都合がよいことを学んだ．すなわち，基底型 T の列（リスト）は，

1. 空列（空リスト）または，
2. T の要素と基底型 T の列（リスト）の連接（連結）

である．

これが，列あるいは繰返しという構成原理を定義する際の道具として，再帰が用いられる理由である．列や繰返しは非常に一般的であり，構造や動作の基本的パターンと考えられるのが普通であるが，それらが再帰によって定義しうることは覚えておくべきである．一方，この逆は真ではない．なぜなら，再帰によってもっと複雑な構造を効果的かつエレガントに定義できるからである．

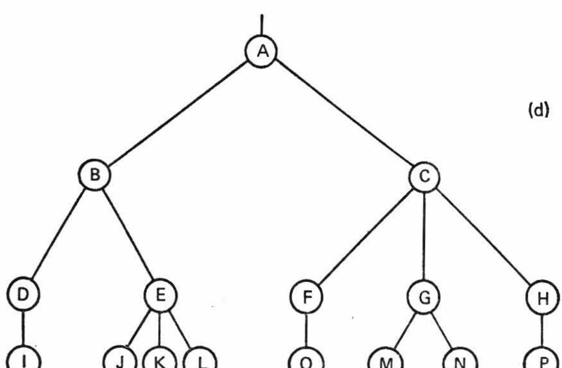


(a)

(A (B (D (I), E (J, K, L)), C (F (O), G (M, N), H (P))))　(b)

```
A
  B
    D
      I
    E
      J
      K
      L
  C
    F
      O
    G
      M
      N
    H
      P
```

(c)



(d)

図 4.17　木構造の表現

(a) 集合の入れ子；(b) 括弧の入れ子；(c) インデンテーション†；(d) グラフ

† **訳注**　indentation. 行の頭を下げる(右にずらす)こと.

木はそのような良く知られた例である．木を次のように定義しよう：基底型 T の**木構造**は，

1. 空構造または，
2. 共通部分のない基底型 T の有限個の木構造（これを**部分木**という）をもった型 T の節点

である．

列と木構造の再帰的定義の類似性から，列（リスト）は，各節点が高々1つの"部分木"しかもたない木構造であることは明らかである．それゆえ，列（リスト）は**縮退した木**ともいわれる．

木構造を表現するには色々の方法がある．たとえば基底型 T が文字からなるとしよう．**図 4.17** には，このような木がいくつかの方法によって示されている．これらの表現はすべて同一の構造を表わしており，したがって，等価である．枝分れ構造を陽に表わしているのはグラフ構造であるが，これが"木"という名前が一般に使用される理由である．全くおかしなことであるが，上下さかさまに，あるいは別ないい方をすれば，木の根を見せるように描くのが普通である．しかしながら，後者のいい方は誤解を招きやすい．なぜなら，通常は最上の節点(A)を**根**と呼ぶからである．自然にある木は，われわれの抽象化よりいくらか複雑ではあるが，これ以後木構造を単に**木**と呼ぶことにする．

順序木は，各節点から出る枝に順序がついている木である．したがって，**図 4.18** の2つの順序木は相異なるものである．節点 x の直接下にある節点 y は，x の（直接の）**子孫** (descendant) といい，x の**レベル**が i であるならば，y のレベルは $i+1$ である．逆に，節点 x は y の（直接の）**先祖** (ancestor) である．木の根はレベル1であると定義される．木の要素の最大レベルを**深さ**または**高さ**という．

子孫を持たない要素は，**端点**または**葉**と呼ばれる．また端点でない要素は，**内点**と呼ばれる．ある内点の（直接の）子孫の数をその節点の**位数**という．またすべての節点についての位数の最大値を木の**位数**という．根からある節点 x に

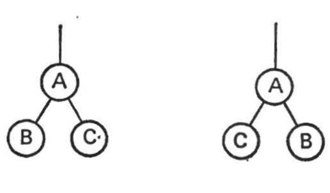


図 4.18　2つの異なった2分木

到達するためにたどる枝の数を，x の**パス長**という†．根のパス長は1であり，その直接の子孫のパス長は2である．一般にレベル i の節点のパス長は i である．木のパス長は，そのすべての節点のパス長の和と定義される．それは，また**内部パス長**とも呼ばれる．たとえば**図4.17**に示されている木の内部パス長は，52である．明らかに，パス長の平均値 P_I は，

$$P_I = \frac{1}{n}\sum_i n_i \cdot i \tag{4.34}$$

で与えられるが，ここで n_i はレベル i の節点の個数である．外部パス長と呼ばれるものを定義するために，木の中で空の部分木が存在するところすべてに特別な節点を付加してその木を拡張する．これを行なう場合，すべての節点は同一の位数，すなわち木の位数をもつと仮定する．したがって，木をこのように拡張することは，子孫を持たない特別な節点によって，空な枝を充たすことに相当する．図4.17の木を特別な節点によって拡張したものが，**図4.19**に示されているが，特別な節点は4角の箱で表わされている．

さて，外部パス長は，すべての特別な節点に関するパス長の和と定義される．レベル i の特別な節点の数を m_i とすると，外部パス長の平均値 P_E は次式で与えられる．

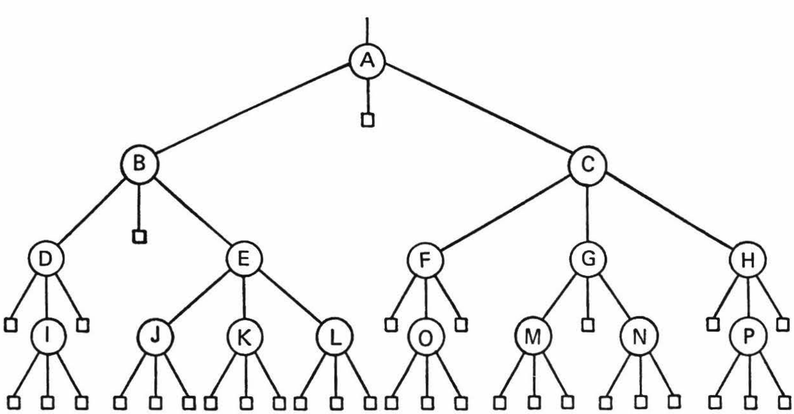


図4.19　特別な節点を使って拡張された3分木

† **訳注**　本書ではグラフによって木を表現する場合，通常のものとは少し異なったものを用いている．すなわち，根に入射する仮想的枝が1本加えられている．パス長の定義では，この仮想的枝も数えられている．

$$P_E = \frac{1}{m}\sum_i m_i \cdot i \tag{4.35}$$

図 4.19 で示された木の外部パス長は 153 である．

位数 d の木に加えられるべき特別な節点の数 m は，もとの節点の数 n に直接依存する．どの節点もそれに入射するただ 1 本の枝をもつことに注意しよう．したがって拡張された木には $m+n$ 本の枝がある．一方，もとの節点からは d 本の枝が出ているが，特別な節点から出る枝はない．ゆえに，$dn+1$ 本の枝が存在することになる．ここで，1 は根に入射する枝によるものである．この 2 つの結果から，特別な節点の数 m ともとの節点の数 n の間の関係として，$dn+1 = m+n$，あるいは次のものが得られる．

$$m = (d-1)n+1 \tag{4.36}$$

与えられた高さ h の木における節点数の最大値は，部分木を持たないレベル h の節点を除いたすべての節点が d 個の部分木を持つときに得られる．位数 d の木に対しては，レベル 1 には 1 つの節点（すなわち根）が含まれ，レベル 2 にはその d 個の子孫が，またレベル 3 には d 個のレベル 2 の節点に対する d^2 個の子孫が含まれ，・・・というようになる．したがって，高さ h，位数 d の木に対する節点数の最大値として次式が得られる．

$$N_d(h) = 1+d+d^2+\cdots+d^{h-1} = \sum_{i=0}^{h-1} d^i \tag{4.37}$$

$d=2$ に対しては，

$$N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h-1 \tag{4.38}$$

特に重要なのは，位数 2 の順序木である．これは，**2 分木**とよばれる．順序 2 分木は，要素（節点）の有限集合であり，空であるか，または 1 つの根(節点)とその**左部分木**および**右部分木**と呼ばれる 2 つの互に素な 2 分木から構成されるものと定義される．以下の節では，特に 2 分木のみを扱うので，語“木”は“順序 2 分木”を意味するものとする． 2 より大きい位数の木は**多分木**と呼ばれるが，これは本章の 5 節で議論される．

2 分木のよく知られた例としては，(i) ある人の父および母を彼の先祖 (!) とするような家系木，(ii) テニスの試合をその勝者を示す節点で表わし，各選手の直前の試合をその先祖とするようなテニス・トーナメントの結果の木，

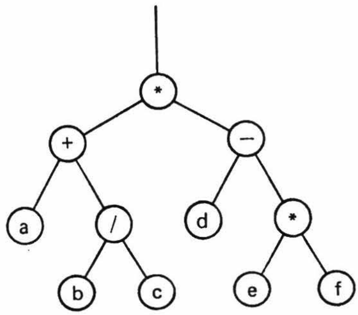


図 4.20　式 $(a+b/c)*(d-e*f)$ の木表現

(iii) 2項演算子を節点とし，オペランドをその部分木とするように算術式を表現した木，等がある(**図 4.20** 参照)．

次に木を表現する問題について考えよう．このような再帰的な構造が枝分れ構造によって図示されることから，ポインタの利用がすぐに思いつく．一定の木構造をその値とするような変数は，明らかに定義する必要がない．その代わり，われわれは，**節点**をある一定の構造をもった変数，すなわち，一定の型の変数として定義する．このとき，その節点の部分木を指すポインタ成分の数は，木の位数によって決定される．明らかに，空な木への参照は **nil** によって示される．ゆえに，図 4.20 の木は次のような型の成分から成り，**図 4.21** のように構成することができる．

```
type node = record op: char;
                   left,right: ↑node
            end
```
(4.39)

明らかに，配列のような他のデータ型を用いても，木構造の抽象的な概念を表現する方法はある．これは，成分を動的に割り付けたり，ポインタを用いてそれを参照するような機能のない言語すべてに共通である．この場合，図 4.20 の木は，

```
t: array[1 .. 11] of
     record op: char;
          left,right: integer
     end
```
(4.40)

のように宣言される配列変数によって，次の**表 4.3** のような値によって表現す

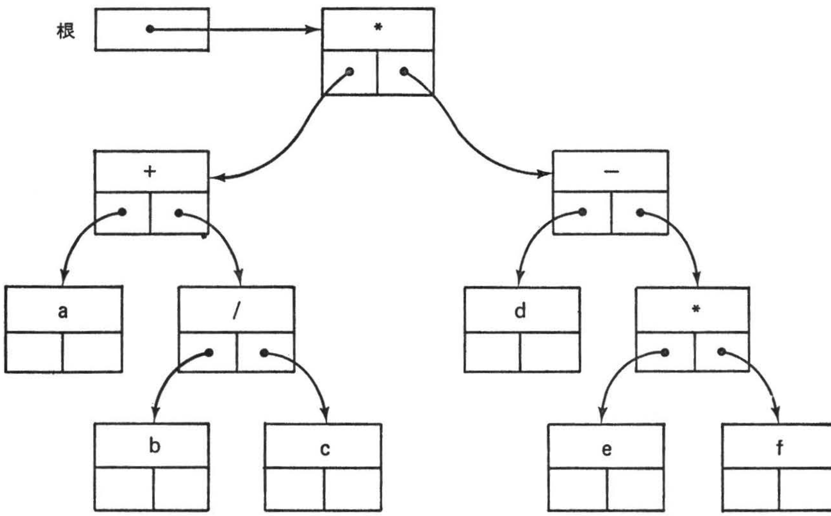


図 4. 21 データ構造として表現された木

ることができる.

配列 t によって表現されるデータの基本的，抽象的構造は木であるが，われわれはこの構造を木とはよばずに，その表面的な宣言に従って配列と呼ぶことにする．また，動的割付け機能のないシステムでの木表現の他の方法についてこれ以上の議論は行なわない．なぜなら，この機能を持ったプログラミング・システムや言語は，現在，あるいは将来一般的に利用可能となると考えられる

1	*	2	3
2	+	6	4
3	−	9	5
4	/	7	8
5	*	10	11
6	*a*	0	0
7	*b*	0	0
8	*c*	0	0
9	*d*	0	0
10	*e*	0	0
11	*f*	0	0

表 4. 3 配列によって表現された木

からである．

木の上手な利用法や操作法について調べる前に，ひとつの木がプログラムによってどのようにして生成されるかという実例を述べる．生成すべき木は，(4.39)で定義された型の節点をもち，入力ファイルから読まれた n 個の数を各節点の値とすると仮定する．また，問題を面白くするために，n 個の節点をもつ高さ最小の木を作ることにする．

与えられた節点数に対して，高さを最小にするためには，最下レベルを除いたすべてのレベルに対して，可能な最大の数の節点を割り付けなければならない．これは，各節点において，入力される節点を左側と右側に等しく分配することによって実現されることは明らかである．このことから，$n=1,\dots,7$ に対しては，図4.22に示されるような木が構成される．

与えられた節点数 n のもとでの等分配の原則は，次のように再帰的に定式化するのが最もよい：

1. 根のために1つの節点を用いる．
2. $nl = n$ **div** 2 個の節点を持つ左部分木をこの方法で生成する．
3. $nr = n-nl-1$ 個の節点を持つ右部分木をこの方法で生成する．

この規則は，**プログラム4.3**において再帰的手続きによって表現されているが，

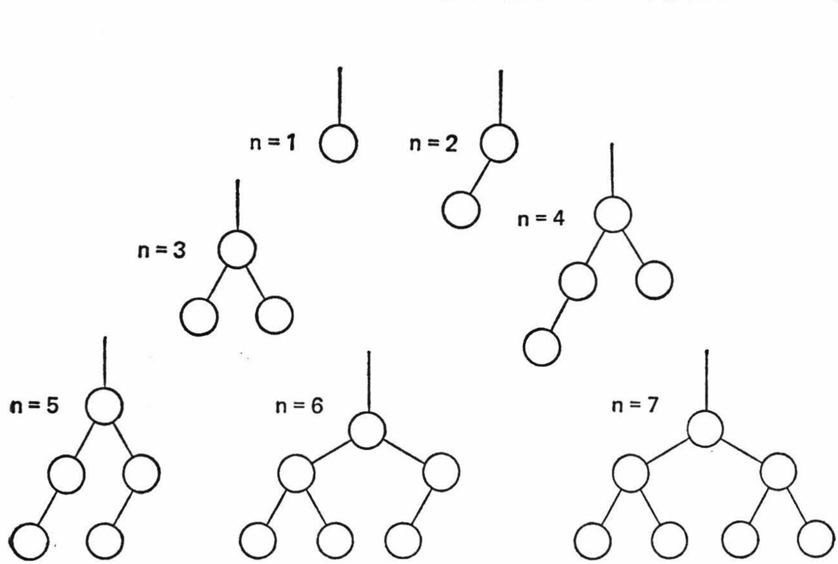


図4.22　完全にバランスした木

プログラム 4.3 完全にバランスした木の構成

```
program buildtree(input,output);
type ref = ↑node;
   node = record key: integer;
                 left, right: ref
          end ;
var n: integer; root: ref;
function tree(n: integer): ref;
   var newnode: ref;
       x, nl, nr: integer;
begin {construct perfectly balanced tree with n nodes}
   if n = 0 then tree := nil else
   begin nl := n div 2; nr := n-nl-1;
      read(x); new(newnode);
      with newnode↑ do
         begin key := x; left := tree(nl); right := tree(nr)
         end ;
      tree := newnode
   end
end {tree} ;
procedure printtree(t: ref; h: integer);
   var i: integer;
begin {print tree t with indentation h}
   if t ≠ nil then
   with t↑ do
   begin printtree(left, h+1);
      for i := 1 to h do write('     ');
      writeln(key);
      printtree(right, h+1)
   end
end {printtree} ;
begin {first integer is number of nodes}
   read(n);
   root := tree(n);
   printtree(root,0)
end .
```

このプログラムは入力ファイルを読み，完全にバランスした木を構成する．ここで，次の定義に注意してほしい．

> 木が**完全にバランスしている**とは，各節点において，その左および右部分木中の節点の数が高々1つしか違わないことである．

21個の節点をもつ木に対する入力データとして，たとえば次のものが与えられたと仮定しよう．

8　9　11　15　19　20　21　7　3　2　1　5　6　4　13　14
10　12　17　16　18

このとき，プログラム4.3は，**図4.23**に示される完全にバランスした木を構成する．

このプログラムの簡潔さと透明さは，再帰的手続きの使用によって得られたものであることに注目してほしい．再帰的アルゴリズムが特に適しているのは，それ自身再帰的に定義された構造をもった情報を処理する場合であることは明らかである．これは，結果の木を印刷する手続きの場合においても，また明白である：空な木に対しては何も印刷されない．レベル L の部分木に対しては，その左部分木をまず印刷し，次に，L 個の空白文字の後にその節点を印刷し，最後に右部分木を印刷する．

再帰的アルゴリズムの有利さは，それを非再帰的形式のものと比較することにより特に明らかになる．読者は(4.41)を見る前に，是非，この木生成プログ

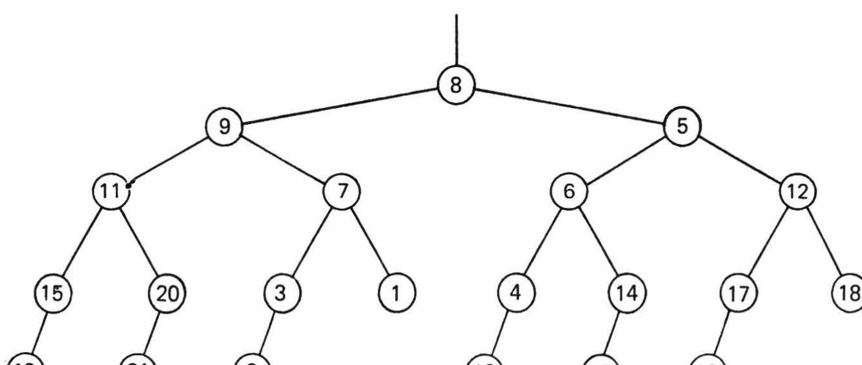


図4.23　プログラム4.3によって生成された木

ラムと等価な非再帰的プログラムを自ら書いてほしい．このプログラムはコメントをつけずに載せてあるが，これがどのように，そして，なぜ動作するのかを理解することは，読者に対する力だめしとして役立つであろう．

```
program buildtree(input,output);
type ref = ↑node;
   node = record key: integer;
                left, right: ref
             end ;
var i,n,nl,nr,x: integer;
   root,p,q,r,dmy: ref;
   s: array [1 . . 30] of  {stack}
          record n: integer; rf: ref
          end ;

begin {first integer is number of nodes}
   read(n); new(root); new(dmy);  {dummy}
   i := 1; s[1].n := n; s[1].rf := root;
   repeat n := s[i] .n; r := s[i] .rf; i := i-1; {pop}          (4.41)
      if n = 0 then r↑.right := nil else
      begin p := dmy;
         repeat nl := n div 2; nr := n-nl-1;
            read(x); new(q); q↑.key := x;
            i := i+1; s[i].n := nr; s[i].rf := q; {push}
            n := nl; p↑.left := q; p := q
         until n = 0;
         q↑.left := nil; r↑.right := dmy↑.left
      end
   until i = 0;
   printtree (root↑.right,0)
end
```

4.4.2 2分木に対する基本操作

木構造に対して実行すべき多くの操作があるが，そのうちでもよく実行されるものに，与えられた操作 P を木の各要素に対して施す，というものがある．このとき，P はすべての節点を訪問するというさらに一般的な作業のパラメータと考えることができるが，この作業は通常**木の走査** (tree traversal) と呼ば

れる．

この作業を1つの順次的プロセスと考えたときには，個々の節点はある特定の順序で訪問され，直線的に配置されると考えることができる．実際，ある基本となる順序にもとづいて，木の中の**次の**要素を指定できれば，アルゴリズムの記述上著しく便利なことが多い．

木構造から自然に生まれた3つの基本的順序がある．木構造の定義と同様に，それらは再帰的に定義するのが便利である．**図4.24**の木についてそれらを述べると次のようになるが，ここで R は根，A, B は左，右部分木を表わすとする．

1. **先順**(preorder)　: R, A, B (部分木より**先**に根を訪問する)
2. **中順**(inorder)　: A, R, B
3. **後順**(postorder)　: A, B, R (部分木の**後**に根を訪問する)

図4.20の木を走査し，節点中の文字を出合った順に記録すると次の列が得られる．

1. 先順：　$* + a / b\ c - d * e f$
2. 中順：　$a + b / c * d - e * f$
3. 後順：　$a\ b\ c / + d e f * - *$

これから，式についての3つの形式が得られる：式の木を先順に走査することにより，**前置記法** (prefix notation) が生じ，後順に走査することにより**後置記法** (postfix notation) が生じる．また，中順の走査からは，**中置記法** (infix notation) が生じるが，演算子の優先順位を示すための括弧は含んでいない．

次に，3つの走査方法を具体的なプログラムによって定式化する．これらのプログラムは，操作されるべき木を表わす陽なパラメータ t と，各節点に対して実行すべき操作を表わす陰なパラメータ P を持っている．次の定義を仮定

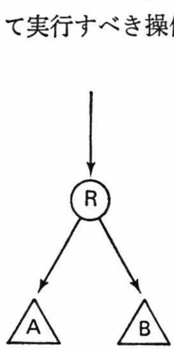


図4.24　2 分 木

する：

```
type ref = ↑node
   node = record ...
              left,right: ref
          end                                    (4.42)
```

3つの方法は，再帰的手続きとして容易に定式化される：再帰的に定義されたデータ構造上の操作は，再帰的アルゴリズムによって記述するのが最も都合がよいことをこれらの手続きは再び示している．

```
procedure preorder(t: ref);
begin if t ≠ nil then
      begin P(t);
            preorder(t↑.left);                   (4.43)
            preorder(t↑.right)
      end
end

procedure inorder(t: ref);
begin if t ≠ nil then
      begin inorder(t↑.left);
            P(t);                                (4.44)
            inorder(t↑.right)
      end
end

procedure postorder(t: ref);
begin if t ≠ nil then
      begin postorder(t↑.left);
            postorder(t↑.right);                 (4.45)
            P(t)
      end
end
```

ポインタ t が値パラメータとして渡されていることに注意せよ．このことは，関係があるのは考えている部分木への**参照**であり，ポインタを値とする変数（この場合には t は変数パラメータとして渡されなければならない）では**ない**ことを示している．

木の走査ルーチンの例としては，各節点のレベルを示すためにインデンテーションをつけて木を印刷するルーチン(プログラム 4.3)がある.

2分木は，一意なキーによる要素の検索が行なわれるデータ集合を表現するためによく用いられる．もし，木の任意の節点 t_i に対して，t_i の左部分木中のすべてのキーが t_i のキーより小さく，t_i の右部分木中のすべてのキーが t_i のキーより大きいならば，この木は**探索木**と呼ばれる．探索木において任意のキーの検索を行なうには，まず根から探索を開始し，各節点において，その左または右部分木の選択をその節点のキーの検査のみによって行ないながら，探索パスを進んでゆけばよい．既に述べたように，n 個の要素からなる集合は，高さが $\log n$ の2分木として構成できる．したがって，もし木が完全にバランスしていれば，$\log n$ 回だけの比較によって，n 個の要素中の探索を行なうことができる．明らかに，このようなデータ集合の表現法としては，木は前節の線形リストより適している.

この探索では，根から目的の節点へのただ1本のパスをたどっているだけであるので，繰返しを用いて容易にプログラムを構成することができる.

```
function loc(x: integer; t: ref): ref;
    var found: boolean;
begin found := false;
    while (t ≠ nil) ∧ ¬found do
    begin
        if t↑.key = x then found := true else
        if t↑.key > x then t := t↑.left else t := t↑.right
    end;
    loc := t
end
```

(4. 46)

値 x をもつキーが t を根とする木の中に見つからない場合には，関数 $loc(x, t)$ は値 **nil** を持つ．リストによる探索の場合と同様に，停止条件の簡単なより良い解をみつけることを考えよう．リストの場合には，それはリストの終わりに番人を立てることであったが，この手法は木の場合にも適用可能である．ポインタを使うことにより，木のすべての枝が同一の番人で終端することが可能である．その結果として生じる構造は，もはや純粋な木ではなく，そのすべての葉が1つの係留点につながれた木となる (**図 4. 25**)．番人は，もとの木を拡張する場合に用いた共通の代表外部節点と考えることができる (図 4. 19 参照)．簡単化された探索ルーチンは (4. 47) のようになる.

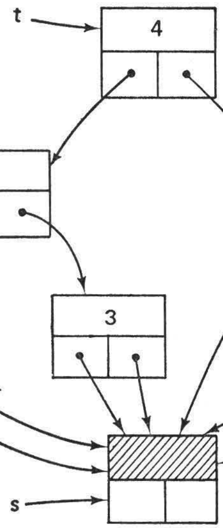


図 4.25　番人のいる探索木

```
function loc(x: integer; t: ref): ref;
begin s↑.key := x;  {sentinel}
   while t↑.key ≠ x do
      if x < t↑.key then t := t↑.left else t := t↑.right;          (4.47)
   loc := t
end
```

このとき，値 x をもつキーが t を根とする木の中で見つけられないならば，*loc* (x, t) の値は s となる．s は **nil** ポインタの役割りをしているだけである．

4.4.3　木の探索と挿入

ポインタによるアクセス機構を用いた動的割付け技法は，データ集合が一度構成されてしまえば以後不変に保たれるような実例によっては，その全能力を示すことはほとんど不可能である．より適切な実例は，プログラムの実行中に木の構造自身が変化する，すなわち成長したり縮んだりするような場合である．この場合には，配列のような他のデータ表現ではうまくゆかず，ポインタによって要素の結合された木が適切な解となる．

まず初めに，定常的に成長し決して縮まないような木の場合についてだけ考えよう．このような例としては，既に連結リストとの関連で取り上げた用語索

引の問題がある．ここでは再びこの問題を考える．この問題では，語からなる列が与えられ，各語の出現回数を決定しなければならない．これは――空な木から出発して――木の中で語の探索を行なうことを意味する．もしそれがあれば，その出現回数は増やされ，無ければ新しい語（回数が1に初期化された）が挿入される．われわれはその作業を，**挿入を伴なう木の探索**と呼ぶ．次のデータ型の定義を仮定する．

```
type ref = ↑word;
     word = record
                 key: integer;
                 count: integer;                    (4.48)
                 left, right: ref
             end
```

さらに，キーよりなる入力ファイル f と，探索木の根を表わす変数 *root* を仮定し，プログラムを次のように定式化する．

```
reset(f);
while ¬eof(f) do                                   (4.49)
   begin read(f, x); search(x,root) end
```

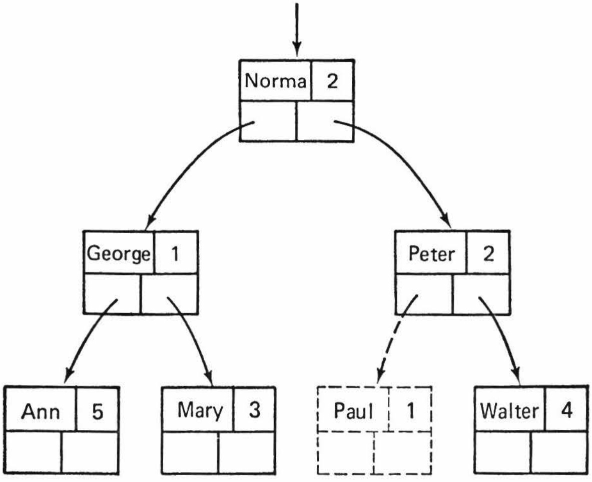


図 4.26　順序2分木への挿入

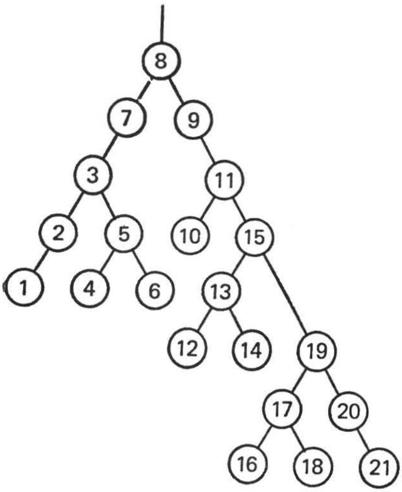


図 4.27 プログラム 4.4 によって生成された探索木

探索パスの発見は再び簡単である．しかしながら，もし“行き止まり”(ポインタ値 **nil** によって示される空な部分木）に達したときには，その空な部分木のところに，与えられた語を挿入しなければならない．たとえば，**図 4.26** で示される 2 分木を考え，語 “Paul” を挿入するとしよう．その結果は，同じ図中に点線で示してある．

操作全体は**プログラム 4.4** に与えられているが，探索過程は再帰的手続きとして定式化されている．パラメータ *p* は変数パラメータであり，値パラメータではないことに注意せよ．これは本質的である．なぜなら，挿入を行なう場合には，それ以前には値 **nil** を持っていたこの変数に，新しいポインタ値を代入しなければならないからである．プログラム 4.3 に入力して図 4.23 を得たのと同じ 21 個の数の列をプログラム 4.4 に加えると，**図 4.27** の 2 分探索木が得られる．

プログラム 4.4　木の探索および挿入

```
program treesearch(input,output);
{binary tree search and insertion}
type ref = ↑word;
    word = record key: integer;
                  count: integer;
                  left, right: ref;
           end ;
```

```
var root: ref; k: integer;

procedure printtree(w: ref; l: integer);
   var i: integer;
begin if w ≠ nil then
      with w↑ do
      begin printtree(left, l+1);
            for i := 1 to l do write('      ');
            writeln(key);
            printtree(right, l+1)
      end
end ;

procedure search(x: integer; var p: ref);
begin
   if p = nil then
   begin {word is not in tree; insert it}
      new (p);
      with p↑ do
         begin key := x; count := 1; left := nil; right := nil
         end
   end else
   if x < p↑.key then search(x, p↑.left) else
   if x > p↑.key then search(x, p↑.right) else
      p↑.count := p↑.count + 1
end {search} ;

begin root := nil;
   while ¬eof(input) do
      begin read(k); search(k, root)
      end ;
   printtree(root,0)
end
```

番人の使用によりこのプログラムは更にいくらか簡単になるが，これを(4.50)に示す．明らかに，プログラムの先頭では，変数 *root* は **nil** ではなく番人へのポインタによって初期化されなければならない．また，各探索の前に，探索すべき値 x を番人のキー・フィールドに代入しなければならない．

```
procedure search(x: integer; var p: ref);
begin
   if x < p↑.key then search(x,p↑.left) else
   if x > p↑.key then search(x,p↑.right) else
   if p ≠ s then p↑.count := p↑.count + 1 else
     begin {insert} new(p);                                    (4.50)
        with p↑ do
        begin key := x; left := s; right := s; count := 1
        end
     end
end
```

再び，そしてこれで最後であるが，再帰を使用しないでこのプログラムを作ってみよう．この場合，再帰の回避は，挿入がない場合のように自明というわけには行かない．なぜなら，挿入が行なわれる場合には，少なくとも1ステップ前だけは探索パスを記憶しておかなければならないからである．プログラム4.4では，この記憶作業は1つの変数パラメータの使用によって自動的に行なわれていた．

```
procedure search(x: integer; root: ref);
   var p1,p2: ref; d: integer;
begin p2 := root; p1 := p2↑.right; d := 1;
   while (p1≠nil) ∧ (d≠0) do
   begin p2 := p1;
      if  x < p1↑.key then
         begin p1 := p1↑.left; d := -1 end else
      if  x > p1↑.key then                                     (4.51)
         begin p1 := p1↑.right; d := 1 end else
      d := 0
   end ;
   if d = 0 then p1↑.count := p1↑.count + 1 else
      begin {insert} new(p1);
         with p1↑ do
         begin key := x; left := nil; right := nil; count := 1
         end ;
         if d < 0 then p2↑.left := p1 else p2↑.right := p1
      end
end
```

挿入すべき要素を正しく結合するためには，その先祖へのポインタと，挿入がその先祖の左または右部分木のどちらに対して行なわれるかを知っていなければならない．このために，2つの変数 $p2$ と d（方向のため）が導入されている．
リストの探索と挿入の場合と同様に，探索パスの走査のために2つのポインタ $p1$ と $p2$ を使用しているが，$p2$ は常に $p1\uparrow$ の先祖を指している．探索プロセスの開始時点でもこの条件が満足されるように，補助的なダミー要素を1つ導入しポインタ *root* によってこれを指す．実際の探索木の原点は，ポインタ *root*↑.*right* によって指される．したがって，プログラムは，もとの代入文

root := **nil**

の代わりに，次の文で始まらなければならない．

new(*root*); *root*↑.*right* := **nil**

このアルゴリズムの目的は探索であるが，ソートにも使うことができる．実際，これは挿入法によるソートに極めて似ている．ただし，配列の代わりに木構造を用いているので，挿入点より上の要素を移動させる必要がない．木を用いたソートは，配列によるソートの最良のものと同程度に能率がよいようにプログラムすることができる．しかし，2, 3の注意が必要である．明らかに，同一のキーが見つかった場合の処理を変えなければならない．もし，$x = p\uparrow.key$ の場合を $x > p\uparrow.key$ の場合と全く同様に処理すれば，これは安定なソート・アルゴリズムになる．すなわち，木を通常の順序（訳註：中順）に走査したとき，同一のキーをもつ項目は挿入されたときと同じ順番に現われる．

一般に，ソートにはもっと良い方法もあるが，探索とソートの両方が必要な場合には，この木の探索と挿入アルゴリズムは強く推薦できるものである．実際，これは，コンパイラやデータ・バンクにおいてデータの蓄積と検索に非常によく利用される方法である．与えられたテキストに対する**相互参照索引** (cross reference index) を作る問題はこの方法の適切な例題であり，以下ではこれを詳細に扱う．

行なうべき作業は，（テキスト f を読み，それに一連行番号をつけて印刷しながら）このテキスト中のすべての語を集め，各語の現われた行番号を記録するプログラムを作ることである．また，この走査の終了後，集められたすべての語とその出現リストをアルファベット順に並べた表を作成する．

明らかに，探索木（辞書式木ともいう）は，テキスト中に現われる語を表現す

る方法としては最も適した方法である．この場合，各節点にはキーのみでなく行番号からなるリストの先頭が含まれる．出現を記録したものを**項目** (*item*) とよぶことにする．したがって，この例題には木と線形リストの両方が含まれる．プログラムは2つの主要部すなわち走査フェーズと表印刷フェーズから成る(**プログラム 4.5** 参照)．後者は木走査ルーチンの簡単な応用であり，節点を訪問する際，キー値（語）の印刷および関連した行番号(項目)リストの走査を行なう．以下はプログラム 4.5“相互参照の生成”に関する明細である．

1. 語は，英字で始まる任意の英数字列であると仮定する．
2. 最初の $c1$ 文字だけがキーとして記録される．したがって，最初の $c1$ 文字では異なっていない2つの語は，同一とみなされる．
3. $c1$ 個の文字は，配列 *id* (型は *alfa*) にパックされる．$c1$ が十分に小さければ，多くの計算機ではこのようなパックされた配列の比較を1命令で行なうことができる．
4. 変数 $k1$ は，文字バッファ a に関する次の不変条件を満たすインデックスとして使用されている．

$$a[i] = '\ ' \quad ただし\ i = k1+1,\ \cdots,\ c1$$

   $c1$ より少ない文字からなる語は，適当な数の空白記号で拡張される．
5. 相互参照索引では，行番号を昇順に印刷することが望ましい．したがって，項目リストは，それが印刷されるのと同じ順に生成されなければならない．このために，リスト中の最初の項目を参照するための2つのポインタが各々の語節点で使われている．
6. 引用符やコメント中の語が索引から除かれるように走査部は作られているが，これらの引用符やコメントは，行の終わりを越えないと仮定している．

プログラム 4.5 相互参照の生成

```
program crossref (f,output);
{cross reference generator using binary tree}
const c1 = 10;     {length of words}
      c2 =  8;     {numbers per line}
      c3 =  6;     {digits per number}
      c4 = 9999;   {max line number}
```

```
type alfa = packed array [1 .. c1] of char;
       wordref = ↑word;
       itemref = ↑item;
       word = record key: alfa;
                        first, last: itemref;
                        left, right: wordref
                  end ;
       item = packed record
                        lno: 0 .. c4;
                        next: itemref
                  end ;
var root: wordref;
       k,k1: integer;
       n: integer;                {current line number}
       id: alfa;
       f: text;
       a: array [1 .. c1] of char;

procedure search (var w1: wordref);
   var w: wordref; x: itemref;
begin w := w1;
   if w = nil then
   begin new(w); new(x);
      with w↑ do
      begin key := id; left := nil; right := nil;
         first := x; last := x
      end ;
      x↑.lno := n; x↑.next := nil; w1 := w
   end else
   if id < w↑.key then search(w↑.left) else
   if id > w↑.key then search(w↑.right) else
   begin new(x); x↑.lno := n; x↑.next := nil;
      w↑.last↑.next := x; w↑.last := x
   end
end {search} ;
procedure printtree(w: wordref);
   procedure printword(w: word);
      var l: integer; x: itemref;
   begin write ('   ', w.key);
      x := w.first; l := 0;
```

```
        repeat if l = c2 then
                begin writeln;
                    l := 0; write ('   ':c1+1)
                end ;
            l := l+1; write (x↑.lno:c3); x := x↑.next
        until x = nil;
        writeln
    end {printword} ;
begin if w ≠ nil then
    begin printtree(w↑.left);
        printword(w↑); printtree(w↑.right)
    end
end {printtree} ;

begin root := nil; n := 0; k1 := c1;
    page (output); reset(f);
    while ¬eof (f) do
    begin if n = c4 then n := 0;
        n := n+1; write (n:c3);          {next line}
        write ('  ');
        while ¬eoln (f) do
        begin {scan non-empty line}
            if f↑ in ['A' .. 'Z'] then
            begin k := 0;
                repeat if k < c1 then
                    begin k := k+1; a[k] := f↑;
                    end ;
                    write (f↑); get (f)
                until ¬(f↑ in ['A' .. 'Z','0' .. '9']);
                if k ≥ k1 then k1 := k else
                repeat a[k1] := '  '; k1 := k1−1
                until k1 = k;
                pack (a,1,id); search(root)
            end else
            begin {check for quote or comment}
                if f↑ = '''' then
                    repeat write(f↑); get(f)
                    until f↑ = '''' else
```

```
              if f↑ = '{' then
                 repeat write(f↑); get(f)
                 until f↑ = '}' ;
              write (f↑); get (f)
           end
        end ;
        writeln; get(f)
     end ;
     page(output); printtree(root);
end
```

表4.4に短いプログラム・テキストを処理した結果を示す.

表4.4　プログラム4.5の出力例

```
 1 PROGRAM PERMUTE (OUTPUT);
 2    CONST N = 4;
 3    VAR I: INTEGER;
 4        A: ARRAY [1..N] OF INTEGER;
 5
 6    PROCEDURE PRINT;
 7       VAR I: INTEGER;
 8    BEGIN FOR I := 1 TO N DO WRITE (A[I]:3);
 9       WRITELN
10    END {PRINT} ;
11
12    PROCEDURE PERM (K: INTEGER);
13       VAR I,X: INTEGER;
14    BEGIN
15       IF K = 1 THEN PRINT ELSE
16       BEGIN PERM (K-1);
17          FOR I := 1 TO K-1 DO
18          BEGIN X := A[I]; A[I] := A[K]; A[K] := X;
19             PERM (K-1);
20                X := A[I]; A[I] := A[K]; A[K] := X;
21          END
22       END
23    END {PERM} ;
24
25 BEGIN
26    FOR I := 1 TO N DO A[I] := I;
27    PERM (N)
28 END
```

ARRAY	4							
A	4	8	18	18	18	18	20	20
	20	20	26					
BEGIN	8	14	16	18	25			
CONST	2							
DO	8	17	26					
ELSE	15							
END	10	21	22	23	28			
FOR	8	17	26					
IF	15							
INTEGER	3	4	7	12	13			
I	3	7	8	8	13	17	18	18
	20	20	26	26	26			
K	12	15	16	17	18	18	19	20
	20							
N	2	4	8	26	27			
OF	4							
OUTPUT	1							
PERMUTE	1							
PERM	12	16	19	27				
PRINT	6	15						
PROCEDURE	6	12						
PROGRAM	1							
THEN	15							
TO	8	17	26					
VAR	3	7	13					
WRITELN	9							
WRITE	8							
X	13	18	18	20	20			

4.4.4 木からの削除

さて，次に挿入の逆の問題，すなわち削除について考えよう．行なうべき作業は，キーの整列した木からキー x をもつ節点を削除する，すなわち取り除くアルゴリズムを定義することである．残念なことに，要素の削除は，一般には挿入ほど簡単ではない．削除すべき要素が端点であったり，1つの子孫しか持たないときには簡単であるが，むずかしいのは，2つの子孫をもつ節点の削除である．なぜなら，1つのポインタによっては，2つの方向を指すことはできないからである．この場合，削除すべき要素をその左部分木の最右の要素または右部分木の最左の要素で置き換えればよい．なぜならば，これらの

要素は，高々1つの子孫しか持たないからである．その詳細は，再帰的手続き *delete* (4.52) に示されている．この手続きでは，3つの場合を区別している．

1. x に等しいキーを持つ要素は存在しない．
2. キー x を持つ要素は，高々1つの子孫しかもたない．
3. キー x を持つ要素は，2つの子孫をもつ．

```
procedure delete (x: integer; var p: ref);
   var q: ref;
   procedure del (var r: ref);
   begin if r↑.right ≠ nil then del (r↑.right) else
         begin q↑.key := r↑.key; q↑.count := r↑.count;
            q := r; r := r↑.left
         end
   end ;
begin {delete}                                                    (4.52)
   if p = nil then writeln (' WORD IS NOT IN TREE') else
   if x < p↑.key then delete(x, p↑.left) else
   if x > p↑.key then delete(x, p↑.right) else
   begin {delete p↑}  q := p;
      if q↑.right = nil then p := q↑.left else
      if q↑.left = nil then p := q↑.right else del (q↑.left);
      {dispose(q)}
   end
end {delete}
```

再帰的な補助手続き *del* は，3. の場合にだけ起動される．これは，削除すべき要素 $q\uparrow$ の左部分木の最右のパスに沿って下りてゆき，その左部分木の最右の要素 $r\uparrow$ の関連情報(キーと回数)によって $q\uparrow$ のそれを置き換える．しかる後，$r\uparrow$ は取り去られる．未定義の手続き *dispose* (q) は，*new* (q) の逆と考えてよい．すなわち，後者が新しい成分に対する記憶割付けを行なうのに対し，前者は，$q\uparrow$ によって占有されていた記憶領域は処分可能であり再利用可能であることを計算機システムに知らせるために使用される（記憶資源の再利用）．

手続き (4.52) の機能を説明するために，**図 4.28** を参照しよう．木(a)が与えられたとき，キー 13, 15, 5, 10 を次々に削除すると，木(b)～(e)が得られる．

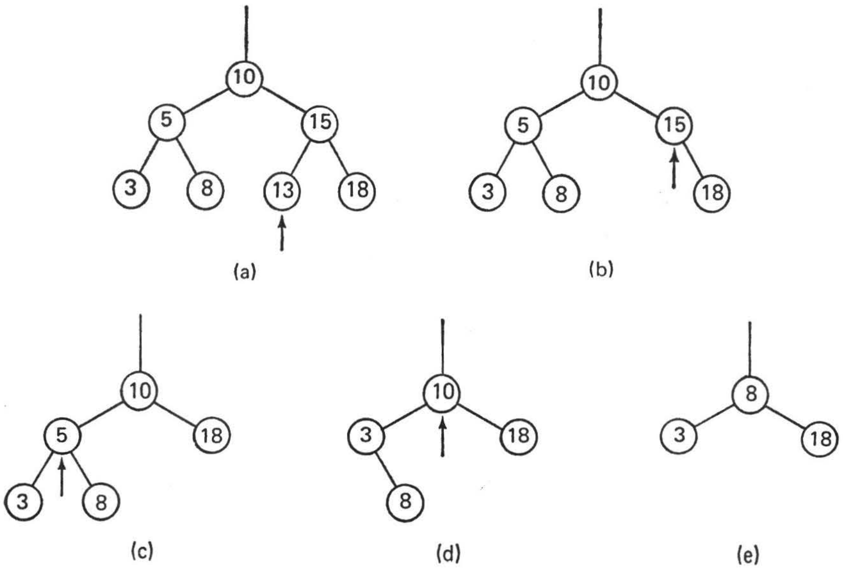


図 4.28 木からの削除

4.4.5 木の探索と挿入の解析

木の探索と挿入アルゴリズムに疑問を感じることは，自然で健康的な反応である．少なくとも，その振舞いについての詳細がもう少し明らかになるまでは，多少の懐疑心はもっているべきである．多くのプログラマが初めに心配することは，木がどのように成長してゆくかを，一般には，われわれが知らないということである：木がどのような形になるかについて，われわれは知っていない．われわれが推論できるのは，それが完全にバランスした木になることは，ほとんどあり得ないことだけである．n 個の節点をもち，完全にバランスした木において，１つのキーを見つけるのに必要な比較の平均回数は，近似的には，$h = \log n$ であるので，このアルゴリズムで生成された木の場合の比較回数は，h より大きいであろう．それでは，どれ位大きいのか．

まず初めに，最悪な場合をみつけるのは簡単である．すべてのキーが，真に昇(または降)順に到着したと仮定する．このときには，各キーはただちに，その子孫の右(左)側に接続され，結果の木は完全に縮退し線形リストになる．し

たがって，探索には平均 $n/2$ 回の比較が必要である．この最悪な場合には，明らかにこのアルゴリズムは極めて性能が悪く，われわれの懐疑心を全面的に正当化することになる．当然，このような場合がどの程度起りやすいかということが疑問として残る．もっと正確には，n 個の相異なるキーの $n!$ 個の置換から生成される $n!$ 個のすべての木と，その n 個のキーすべてに関して，探索パス長の平均値 a_n を知りたい．アルゴリズムの解析に関するこの問題は，かなり簡単であることが判明するが，その結果が実際的に重要であると同時に，アルゴリズム解析の典形的な例であるので以下これについて述べる．

値が $1, 2, \cdots, n$ の相異なる n 個のキーが与えられたとする．最初のキー――これは明らかに根節点になる――が値 i をもつ確率は $1/n$ である．その左部分木は最終的には $i-1$ 個の節点を，また右部分木は $n-i$ 個の節点を含む(**図4.29**参照)．左部分木中の探索パスの平均長を a_{i-1}，右部分木中のそれを a_{n-i} とし，残りの $n-1$ 個のキーの可能なすべての置換は等しく起こりやすいと仮定する．n 個の節点からなる木における平均パス長は，各節点のレベルとそのアクセス確率の積の和である．よって，もしすべての節点が等確率でアクセスされると仮定すると，

$$a_n = \frac{1}{n}\sum_{i=1}^{n} p_i \tag{4.53}$$

である．ただし，p_i は節点 i のパス長である．

図4.29において，節点を3つのクラスに分ける．

1. 左部分木中の $i-1$ 個の節点は，平均パス長 $a_{i-1}+1$ をもつ．
2. 根はパス長1をもつ．
3. 右部分木中の $n-i$ 個の節点は，平均パス長 $a_{n-i}+1$ をもつ．

したがって，(4.53)は3つの項の和として表現することができる．

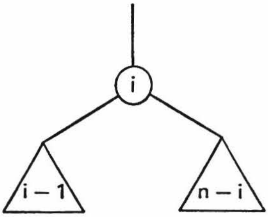


図4.29　枝の重み分布

$$a_n^{(i)} = (a_{i-1}+1)\frac{i-1}{n}+1\cdot\frac{1}{n}+(a_{n-i}+1)\frac{n-i}{n} \qquad (4.54)$$

目的の値 a_n は，すべての $i=1,\cdots,n$ についての，すなわち，根のキーが1, 2,…, n であるすべての木についての $a_n^{(i)}$ の平均値として導かれる．

$$\begin{aligned} a_n &= \frac{1}{n}\sum_{i=1}^{n}\left[(a_{i-1}+1)\frac{i-1}{n}+\frac{1}{n}+(a_{n-1}+1)\frac{n-i}{n}\right] \\ &= 1+\frac{1}{n^2}\sum_{i=1}^{n}[(i-1)a_{i-1}+(n-i)a_{n-i}] \qquad (4.55) \\ &= 1+\frac{2}{n^2}\sum_{i=1}^{n}(i-1)a_{i-1} = 1+\frac{2}{n^2}\sum_{i=1}^{n-1}i\cdot a_i \end{aligned}$$

方程式(4.55)は，a_n に対する再帰関係式であり，$a_n = f_1(a_1, a_2, \cdots, a_{n-1})$ の形をしている．これから，$a_n = f_2(a_{n-1})$ の形をしたより簡単な再帰関係式が以下のように導かれる．

(4.55)から直接次式が得られる．

(1) $a_n = 1+\frac{2}{n^2}\sum_{i=1}^{n-1}i\cdot a_i = 1+\frac{2}{n^2}(n-1)a_{n-1}+\frac{2}{n^2}\sum_{i=1}^{n-2}i\cdot a_i$

(2) $a_{n-1} = 1+\frac{2}{(n-1)^2}\sum_{i=1}^{n-2}i\cdot a_i$

(2)に $(n-1/n)^2$ を掛けると，

(3) $\frac{2}{n^2}\sum_{i=1}^{n-2}i\cdot a_i = \frac{(n-1)^2}{n^2}(a_{n-1}-1)$

(3)を(1)に代入することにより，

$$a_n = \frac{1}{n^2}((n^2-1)a_{n-1}+2n-1) \qquad (4.56)$$

したがって，a_n は調和関数を用いて非再帰的に閉じた形で表現できることがわかる．

$$\begin{aligned} H_n &= 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} \\ a_n &= 2\frac{n+1}{n}H_n-3 \end{aligned} \qquad (4.57)$$

[疑い深い読者は(4.57)が再帰関係式(4.56)を満たすことを確かめてほしい]

オイラーの公式から

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + \cdots$$

ここで $\gamma \cong 0.577$ はオイラー定数である．したがって，大きい n に対しては，次の関係式が導かれる．

$$a_n \cong 2[\ln(n) + \gamma] - 3 = 2\ln(n) - c$$

完全にバランスした木における平均パス長は，近似的には

$$a'_n = \log(n) - 1 \tag{4.58}$$

であるので，大きな n に対しては意味のない定数項を無視すると

$$\lim_{n\to\infty} \frac{a_n}{a'_n} = \frac{2\ln n}{\log n} = 2 \cdot \ln 2 \cong 1.386 \tag{4.59}$$

この解析の結果(4.59)から何を学ぶことができるであろうか．それは，プログラム 4.4 から得られる“ランダム”な木の代わりに，苦労して完全バランス木を常に構成しても，平均パス長を高々39％しか改善できる見込みがない――すべてのキーが等確率で検索されるとした場合であるが――ことを述べている．“平均”という語は強調されるべきである．なぜなら，（挿入すべき n 個のキーの置換が等しく起こりやすいときには）非常にまれなことではあるが，生成された木が完全にリストに縮退してしまうという不運な場合には，もちろん改善は非常に大きいからである．これと関連して，“ランダム”な木の平均パス長も節点数とともに真に対数的にしか増加しないことは注目に値する．もちろん最悪な場合には，パス長は線形に増加する．

この39％という数字は，キーの挿入時に行われる木構造の再編成のための余分な努力に限界を課するものである．あたりまえのことであるが，節点（情報）へのアクセス（検索）の頻度と挿入の頻度の比 r は，そのような企てが引き合う限界に大きな影響を与える．この比が大きければ大きい程，再編成手続きから得られる利益は大きい．ほとんどの応用では39％という数字は小さすぎ，節点数やアクセス対挿入比が大きい（あるいは最悪な場合が心配される）場合を除いては，木への単純な挿入に対する改良は引き合わない．

4.4.6　バランス木

前節の議論から，木の構造を完全なバランス状態に復元させるような挿入手

続きについては，その利用が有利であるような機会はほとんどない．なぜなら，ランダムに挿入した後で完全なバランスを取り戻すには，かなりこみ入った操作が必要だからである．したがって，実行可能な改善法を得るには，“バランス”についてのより緩やかな定義を定式化しなければならない．そのような“不完全”なバランス基準は，平均探索性能が多少低下する代わりに，木再編成手続きがより簡単になるものでなければならない．

バランスに関するそのような1つの定義が，Adelson-Velskii と Landis [4-1] によって与えられた．このバランス基準は次のようである．

> 木が**バランス**しているとは，任意の節点に対して，その2つの部分木の高さが高々1しか違わないことである．

この条件を満たす木は，しばしば（発明者にちなんで）**AVL 木**と呼ばれる．われわれはこのような木を，単に**バランス木**と呼ぶが，これはこのバランス基準が非常に適切なものだからである．（完全にバランスした木は，またAVLバランス条件を満たすことに注意せよ）．

この条件は単に簡単なばかりでなく，再バランス手続きがともかく実現可能であり，また平均探索パス長が完全バランス木と実際的には同程度になる．

次の各操作は，バランス木に対しては，最悪の場合でも $O(\log n)$ 単位時間で実行することができる．

1. 与えられたキーをもつ節点の発見
2. 与えられたキーをもつ節点の挿入
3. 与えられたキーをもつ節点の削除

これらのことは，Adelson-Velskii と Landis によって証明されたある定理の直接の結果であるが，この定理は節点の数がいかに多くても，バランス木は完全バランス木に比べて，その高さが45%以上増えることはないことを保証している．n 個の節点をもつバランス木の高さを $h_b(n)$ とすると，

$$\log(n+1) \le h_b(n) \le 1.4404\log(n+2)-0.328 \qquad (4.60)$$

最適値が得られるのは，もちろん $n=2^k-1$ に対して，木が完全にバランスしているときである．それでは，AVLバランス木が最悪な構造になるのはどのような場合であろうか．

n 個の節点からなるバランス木の高さの最大値 h を見つけるために，h を固定し，最小の節点数のバランス木を構成することを考えよう．この方針が推奨される理由は，h の最小値の場合と同様に，求める値が得られるのが特定の n

の値に対してだけだからである．高さ h のこの木を T_h で表わそう．明らかに T_0 は空木であり，T_1 は節点1つからなる木である．$h>1$ に対して T_h を構成するためには，再び節点数最小の2つの部分木と根を用意すればよい．このとき，この部分木は，また，T_i の形のものである．明らかに，部分木のうち一方はその高さが $h-1$ でなければならず，他方のものは高さが1だけ小さい，すなわち，$h-2$ のものが許される．**図4.30** に高さが2,3,4の場合の木を示す．それらの木の合成規則が，フィボナッチ数に非常によく似ていることから，これらはフィボナッチ木と呼ばれる．その定義は次のようである．

1. 空木は高さ0のフィボナッチ木である．
2. 節点1つからなる木は，高さ1のフィボナッチ木である．
3. T_{h-1}，T_{h-2} を高さ $h-1$，$h-2$ のフィボナッチ木とすると，$T_h=<T_{h-1}, x, T_{h-2}>$ は高さ h のフィボナッチ木である．
4. これ以外のものはフィボナッチ木ではない．

T_h 中の節点の数は，次の簡単な再帰公式によって与えられる．

$$\begin{aligned} N_0 &= 0, \qquad N_1 = 1 \\ N_h &= N_{h-1}+1+N_{h-2} \end{aligned} \tag{4.61}$$

数 N_i は，(4.60)の最悪な場合（h の上限）に対応する節点数である．

4.4.7　バランス木への挿入

次に，バランス木に新しい節点が挿入されるときに，何が起こるかを考えよう．根 r とその左右部分木 L, R が与えられたとき，3つの場合を区別しなければならない．新しい節点が L に挿入されて，その高さが1だけ増加したと仮定する†：

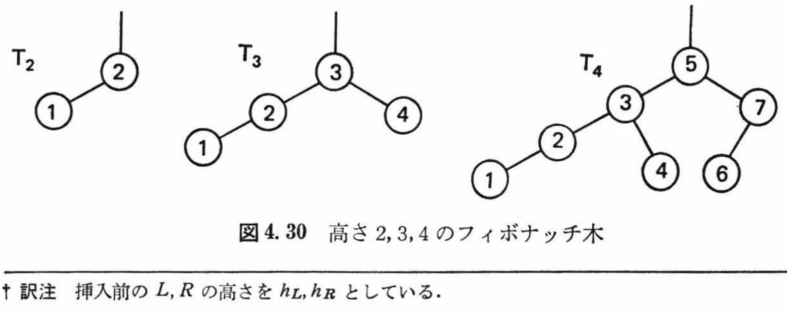


図4.30　高さ2,3,4のフィボナッチ木

† 訳注　挿入前の L, R の高さを h_L, h_R としている．

1. $h_L=h_R$：L と R の高さは等しくなくなるが，バランス基準には違反しない．
2. $h_L<h_R$：L と R は同じ高さになる，すなわちバランスはより一層よくなる．
3. $h_L>h_R$：バランス条件に違反するので，木を再編成しなければならない．

図4.31 の木を考えよう．キーが 9 と 11 の節点は，再バランス化をせずに挿入することができる；その結果，根が 10 の木は一方に偏った形になるであろう（場合**1.**）．また，根が 8 の木はそのバランスを改善するであろう（場合**2.**）．一方，節点 1, 3, 5 または 7 の挿入には，再バランス化が必要である．

色々な状況を注意深く調べることにより，別個の扱いが必要な本質的に異なる場合は，2 つしかないことがわかる．残りのものは，これらの 2 つから対称性を考慮することにより導くことができる．ひとつの場合は，図 4.31 の木へのキー 1 または 3 の挿入により特徴づけられ，また，もうひとつの場合は，キー 5 または 7 の挿入により特徴づけられる．

図4.32 には，一般化されたこの 2 つの場合が示されているが，ここで四角い箱は部分木を表わし，挿入によって増加した高さは，×印により示されている．簡単な変換により，この 2 つの構造は望ましいバランスを取り戻すことができる．その結果が**図4.33** に示されている：許される移動は垂直方向のものだけであり，示されている節点や部分木の相対的水平位置は不変でなければならないことに注目せよ．

挿入と再バランス化のアルゴリズムは，バランスに関する情報の記憶の仕方に非常に依存する．ひとつの極端な解は，バランス情報を木構造自身の形として完全に暗黙的に記憶することである．しかしながら，この場合には，挿入によって変化を受けるごとに節点のバランス度を決定しなければならず，その結果オーバヘッドが極めて大きくなってしまう．もう一方の極端な方法は，各節

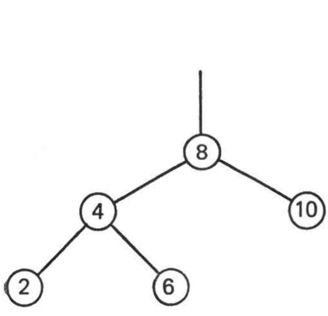


図4.31 バランス木

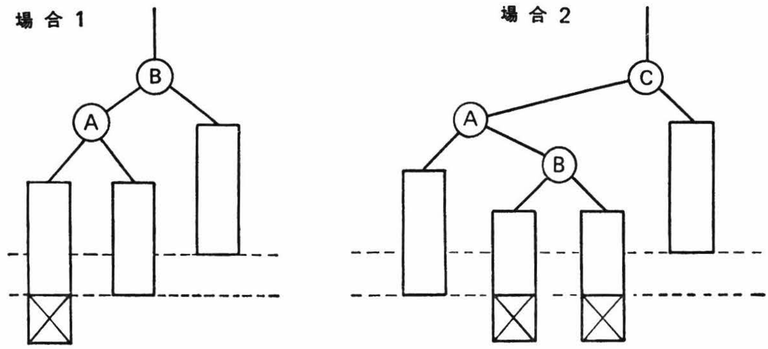


図 4. 32　挿入により生じたアンバランス

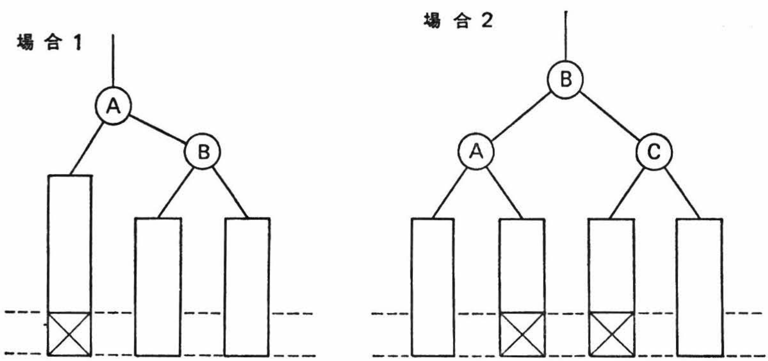


図 4. 33　バランスの復元

点に対してバランス度を陽に付随させるものである．このときには，(4. 48)の節点の定義を次のように拡張する．

$$\begin{aligned}\textbf{type}\ \mathit{node}\ =\ \textbf{record}\ & \mathit{key}\colon \mathit{integer};\\ & \mathit{count}\colon \mathit{integer};\\ & \mathit{left,right}\colon \mathit{ref};\\ & \mathit{bal}\colon -1\,..\,+1\\ \textbf{end}\ & \end{aligned} \tag{4.62}$$

これ以後，節点のバランス度は，その右部分木の高さから左部分木の高さを

引いたものと考え，この節点の定義(4.62)に基づいてアルゴリズムを構成する．

節点の挿入過程は，本質的には次の3つの連続した部分から成っている：

1. そのキーが木の中に存在しないことが確認されるまで探索パスをたどる．
2. 新しい節点を挿入し，その結果のバランス度を決定する．
3. 探索パスに沿って戻りながら，各節点のバランス度を検査する．

この方法には余分な検査が含まれているが（一度バランスが確立されてしまえば，その節点の先祖については検査する必要がない），この方法は，既に得られしている探索と挿入手続き（プログラム4.4）の純粋な拡張として実現することができるので，まずはこの明らかに正しい方式に従っていくことにする．この手続きは，各節点で必要となる探索操作を記述しており，また再帰的な形式をしているので，“探索パスに沿って戻りながら”行なう付加的操作を容易に追加することができる．各ステップにおいては，（挿入が行なわれた）部分木の高さが増加したか否かについての情報が渡されなければならない．したがって，手続きのパラメータ・リストに論理型のパラメータ h を追加するが，これは，“その部分木の高さが増加した”ことを意味している．明らかに，h は結果を伝達するために使用されるので，変数パラメータでなければならない．

いまプログラムの実行が，ある節点 $p\uparrow$ の左部分木から，その高さを増加させたという情報をもって節点 $p\uparrow$ に戻りつつあると仮定する．このとき，挿入前の部分木の高さとの関連で，3つの状況を区別しなければならない：

1. $h_L < h_R$, $p\uparrow.bal = +1$, 挿入前のアンバランスは是正された．
2. $h_L = h_R$, $p\uparrow.bal = 0$, 重みが左側に傾いた．
3. $h_L > h_R$, $p\uparrow.bal = -1$, 再バランス化が必要である．

3番目の場合には，左部分木の根のバランス度（$p1\uparrow.bal$ とする）を調べて，図4.32の場合1または場合2のどちらになっているかを決定する．もし，その節点の左部分木の方が右部分木より高い場合には，場合1として扱い，そうでないときには，場合2として扱う（この場合には，左部分木の根のバランス度が0であることはあり得ないことを読者自ら確認してほしい）．必要となる再バランス化操作は，ポインタの代入の系列として完全に表現することができる．実際，ポインタは巡回的に交換され，その結果，関係する2個あるいは3個の節点に関する単一または2重の回転が行なわれる．ポインタの交換に加え

て，関係する節点のバランス度の調整も行なわなければならない．その詳細は，探索，挿入および再バランス化のための手続き (4.63) に示されている．

```
procedure search(x: integer; var p: ref; var h: boolean);
    var p1,p2: ref;      {h = false}
begin
    if p = nil then
    begin {word is not in tree; insert it}
        new(p); h := true;
        with p↑ do
        begin key := x; count := 1;
            left := nil; right := nil; bal := 0
        end
    end else
    if x < p↑.key then
    begin search(x, p↑.left, h);
        if h then      {left branch has grown higher}
        case p↑.bal of
      1:  begin p↑.bal := 0; h := false
          end ;                                                  (4.63)
      0:  p↑.bal := -1;
     -1:  begin {rebalance} p1 := p↑.left;
              if p1↑.bal = -1 then
              begin {single LL rotation}
                  p↑.left := p1↑.right; p1↑.right := p;
                  p↑.bal := 0; p := p1
              end else
              begin {double LR rotation}  p2 := p1↑.right;
                  p1↑.right := p2↑.left; p2↑.left := p1;
                  p↑.left := p2↑.right; p2↑.right := p;
                  if p2↑.bal = -1 then p↑.bal := +1 else p↑.bal := 0;
                  if p2↑.bal = +1 then p1↑.bal := -1 else p1↑.bal := 0;
                  p := p2
              end ;
              p↑.bal := 0; h := false
          end
      end
    end else
    if x > p↑.key then
```

```
    begin search(x, p↑.right, h);
      if h then     {right branch has grown higher}
      case p↑.bal of
    -1: begin p↑.bal := 0; h := false
        end ;
     0: p↑.bal := +1;
     1: begin {rebalance} p1 := p↑.right;
          if p1↑.bal = +1 then
          begin {single RR rotation}
            p↑.right := p1↑.left; p1↑.left := p;
            p↑.bal := 0; p := p1
          end else
          begin {double RL rotation}  p2 := p1↑.left;
            p1↑.left := p2↑.right; p2↑.right := p1;
            p↑.right := p2↑.left; p2↑.left := p;
            if p2↑.bal = +1 then p↑.bal := -1 else p↑.bal := 0;
            if p2↑.bal = -1 then p1↑.bal := +1 else p1↑.bal := 0;
            p := p2
          end ;
          p↑.bal := 0; h := false
        end
      end
    end
    else
    begin p↑.count := p↑.count + 1; h := false
    end
end {search}
```

その動作原理は**図 4.34** に示されている．2つの節点だけからなる2分木(a)を考えよう．キー7の挿入により，まずアンバランスな木(線形リスト)が生じる．そのバランス化は，**単一** *RR* **回転** (single *RR* rotation) によって行なわれ，完全にバランスした木 (b) が得られる．さらに節点2および1を挿入すると，4を根とする部分木がアンバランスになるが，この部分木は，**単一** *LL* **回転** (single *LL* rotation) によりバランス化され (d) になる．引き続いてキー3を挿入すると，根節点5のバランス条件はくずれるが，これはより複雑な**2重** *LR* **回転** (double *LR* rotation) により再確立される．その結果は木 (e) である．この状態では，次の挿入によりバランスが失われる候補は節点5だけである．

事実節点6の挿入により，(4.63)で述べられている4番目の再バランス化，すなわち**2重** RL **回転**(double RL rotation) が起動される．その結果，最終的な木は(f)になる．

バランス木への挿入アルゴリズムの性能に関して，特に興味のある次の2つの問題がある：

1. もし n 個のキーの $n!$ 個の置換が等確率で起るとすると，構成されたバランス木の高さの期待値はいくらか．
2. 挿入が再バランス化を要求する確率はいくらか．

このこみ入ったアルゴリズムの数学的解析はまだ未解決であるが，実験的テストによれば，(4.63)によって生成されたバランス木の高さの期待値は，$h = \log(n) + c$ であると推論される．ここで，c は小さな定数である($c \cong 0.25$)．したがって，AVL バランス木は，その管理が完全バランス木に比較して非常に簡単であるにもかかわらず，実際には同等の性能をもつことがわかる．さらに，約2回の挿入に対し，平均1回の再バランス化が必要であることを示す実験的証拠もある．このとき，単一および2重の回転は同様に起りやすい．**図4.34** の例題は，明らかに，できるかぎり多くの回転の説明を最小の数の挿入によっ

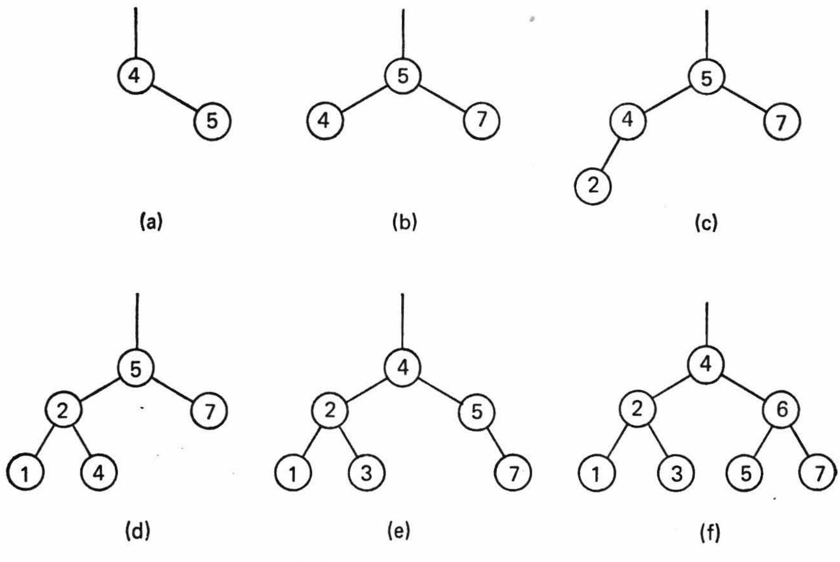


図4.34　バランス木への挿入

て説明するために，注意深く選ばれたものである！

バランス化操作の複雑さは，挿入に比べて検索が著しく頻繁であるような場合にのみバランス木は使用されるべきであることを示唆している．通常そのような探索木は，記憶領域を節約するために密に詰められたレコードとして実現されるので，このことは全く正しい．アクセスおよびバランス度の更新——いずれも2ビットしか必要としないが——の速度は，それゆえに再バランス化操作の効率に対して決定的な要因になることが多い．レコードを密に詰めることが要求される場合には，バランス木はその長所の多くを失うことを実験的評価は示している．木への単純な挿入アルゴリズムを打ち負かすことは実際困難である

4.4.8 バランス木からの削除

木からの削除についての経験から類推すると，バランス木の場合にも，挿入より削除の方がより複雑であることが予想される．実際これは正しいが，再バランス化操作は本質的には挿入の場合と変わりがない．すなわち，再バランス化は節点の単一および2重の回転からなる．

バランス木からの削除のための基礎は，アルゴリズム(4.52)である．端点およびただ1個の子孫をもつ節点の場合には削除は容易である．もし，削除すべき節点が2つの部分木を持っている場合には，われわれはこれを再び左部分木の最右節点で置き換える．挿入(4.63)の場合と同様に，“部分木の高さが減少した”ことを意味する論理型の変数パラメータ *h* を追加し，再バランス化は *h* が *true* のときのみ行なう．節点が見つけられて削除されるか，あるいは再バランス化自身によって部分木の高さが減少したとき，*h* には値 *true* が代入される．(4.64)においては，2つの(対称的な)バランス化操作を手続きの形で導入しているが，この理由は，これが削除アルゴリズムにおいて1個所以上の場所から呼出される必要があるからである．balance 1 は左部分木の高さが減少したときに，また，balance 2 は右部分木の高さが減少した後で適用されることに注意せよ．

```
procedure delete(x: integer; var p: ref; var h: boolean);
   var q: ref;      {h = false}
   procedure balance1(var p: ref; var h: boolean);
      var p1,p2: ref; b1,b2: -1 .. +1;
```

```
begin {h = true, left branch has become less high}
   case p↑.bal of
 -1:  p↑.bal := 0;
  0:  begin p↑.bal := +1; h := false
      end ;
  1:  begin {rebalance} p1 := p↑.right; b1 := p1↑.bal;
          if b1 ≥ 0 then
          begin {single RR rotation}
              p↑.right := p1↑.left; p1↑.left := p;
              if b1 = 0 then
              begin p↑.bal := +1; p1↑.bal := -1; h := false
              end else
              begin p↑.bal := 0; p1↑.bal := 0
              end ;
              p := p1
          end else
          begin {double RL rotation}
              p2 := p1↑.left; b2 := p2↑.bal;
              p1↑.left := p2↑.right; p2↑.right := p1;          (4.64)
              p↑.right := p2↑.left; p2↑.left := p;
              if b2 = +1 then p↑.bal := -1 else p↑.bal := 0;
              if b2 = -1 then p1↑.bal := +1 else p1↑.bal := 0;
              p := p2; p2↑.bal := 0
          end
      end
   end
end {balance 1} ;

procedure balance2(var p: ref; var h: boolean);
   var p1,p2: ref; b1,b2: -1 .. +1;
begin {h = true, right branch has become less high}
   case p↑.bal of
  1:  p↑.bal := 0;
  0:  begin p↑.bal := -1; h := false
      end ;
 -1:  begin {rebalance} p1 := p↑.left; b1 := p1↑.bal;
          if b1 ≤ 0 then
```

```
            begin {single LL rotation}
               p↑.left := p1↑.right; p1↑.right := p;
               if b1 = 0 then
               begin p↑.bal := -1; p1↑.bal := +1; h := false
               end else
               begin p↑.bal := 0; p1↑.bal := 0
               end ;
               p := p1
            end else
            begin {double LR rotation}
               p2 := p1↑.right; b2 := p2↑.bal;
               p1↑.right := p2↑.left; p2↑.left := p1;
               p↑.left := p2↑.right; p2↑.right := p;
               if b2 = -1 then p↑.bal := +1 else p↑.bal := 0;
               if b2 = +1 then p1↑.bal := -1 else p1↑.bal := 0;
               p := p2; p2↑.bal := 0
            end
         end
      end
   end {balance2} ;                                              (4.64)
   procedure del(var r: ref; var h: boolean);
   begin {h = false}
      if r↑.right ≠ nil then
      begin del(r↑.right,h); if h then balance2(r,h)
      end else
      begin q↑.key := r↑.key; q↑.count := r↑.count;
         r := r↑.left; h := true
      end
   end ;

begin {delete}
   if p = nil then
      begin writeln ('KEY IS NOT IN TREE'); h := false
      end else
   if x < p↑.key then
      begin delete(x,p↑.left,h); if h then balance1(p,h)
      end else
   if x > p↑.key then
      begin delete(x,p↑.right,h); if h then balance2(p,h)
      end else
```

```
  begin {delete p↑}  q := p;
     if q↑.right = nil then
        begin p := q↑.left; h := true
        end else
     if q↑.left = nil then
        begin p := q↑.right; h := true
        end else                                   (4.64)
     begin del(q↑.left, h);
        if h then balance1(p, h)
     end ;
     {dispose(q)}
  end
end {delete}
```

この手続きの動作が**図 4.35** に示されている．バランス木(a)が与えられたとき，キーが 4, 8, 6, 5, 2, 1 および 7 の節点を次々に削除してゆくと，木(b), …, (h)が得られる．

キー4は端点であるので，その削除自体は簡単である．しかしながら，その結果として，節点3がアンバランスになる．その再バランス化操作には，単一 *LL* 回転が含まれる．節点6の削除の後でも再バランス化が必要になる．このときには，根(7)の右部分木は単一 *RR* 回転によって再バランス化される．節点2はただ1つの子孫しかもたないので，その削除自身は単純であるが，複雑な2重 *RL* 回転を必要とする．第4番目の場合，すなわち，2重 *LR* 回転は，節点7がその左部分木の最右の要素，すなわちキー3の節点，で置き換えられて除去された後で最後に起動される．

明らかに，バランス木からの1つの要素の削除も，――最悪の場合でも――$O(\log n)$ の操作によって実行可能である．しかしながら，挿入手続きと削除手続き動作の本質的な差を見落としてはならない．キー1個の挿入には，高々1回の（2個または3個の節点に関する）回転しか必要でなかったのに反し，削除の場合には，探索パスに沿った各節点において回転が要求される可能性がある．たとえば，フィボナッチ木の最右の節点を削除する場合を考えよう．この場合，どの節点をひとつ削除しても木の高さは減少する：さらに，その最右の節点の削除には，最も多くの回転を必要とする．したがって，これは最悪なバランス木における最悪な節点の選択であり，不運な偶然の巡り合わせということになる！　それでは，一般には回転はどの程度起こりやすいのであろうか．

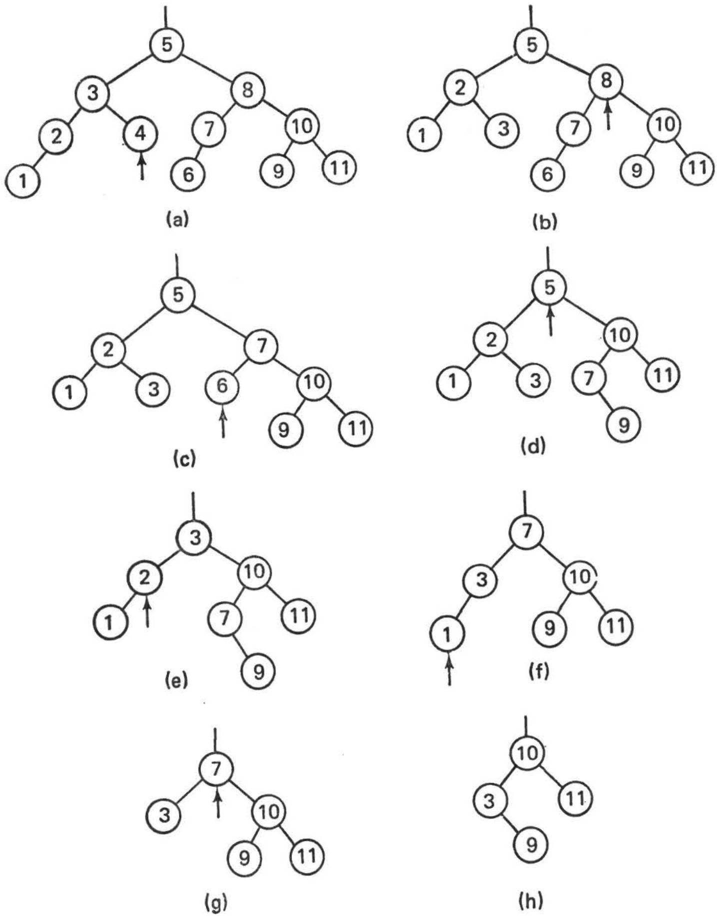


図 4.35 バランス木からの削除

約2回の挿入に対して1回の回転が行なわれるのに反し，5回の削除に対して，回転は1回しか必要ではないという驚くべき実験結果がある．したがって，バランス木からの削除は，挿入と同程度に容易——あるいは複雑——である．

4.4.9 最適探索木

探索木の構成に関するわれわれのこれまでの考察は，アクセスの頻度がすべ

ての節点について等しい，すなわち，すべてのキーが探索の引数として同程度に起こりやすいという仮定にもとづいてきた．これは，アクセスの分布が全く未知の場合には，多分最もよい仮定であろう．しかしながら，場合によっては（これは通常というより，むしろ例外的ではあるが）個々のキーへのアクセスの確率が入手可能なこともある．このような場合には，キーの集合は不変であり，探索木は挿入されることも削除されることもない，すなわち一定の構造を保持するという特徴をもっていることが普通である．典型的な例としては，コンパイラの走査プログラムがあるが，これはソース・プログラム中の各語（識別名）がキー・ワード（予約語）であるか否かを判定するためのものである．この場合には，コンパイルされた数百のプログラムを統計的に計測することにより，各キーの相対的な出現頻度，したがって，アクセス頻度に関する正確な情報を得ることができるであろう．

探索木において，節点 i のアクセスされる確率を p_i とする．

$$Pr\{x = k_i\} = p_i, \quad \sum_{i=1}^{n} p_i = 1 \tag{4.65}$$

このとき，探索ステップの総数——十分に多くの試行について合計した——が最小になるように探索木を構成したい．この目的のために，各節点にある一定の重みをつけ，パス長の定義(4.34)を修正する．頻繁にアクセスされる節点の重みは大きくし，稀にしかアクセスされないような節点の重みは小さくする．このとき，(内部)**重みづけパス長**は，根から各節点へのパス長をその節点のアクセス確率で重みづけし，加えたものである．

$$p_I = \sum_{i=1}^{n} p_i h_i \tag{4.66}$$

ここで，h_i は節点 i のレベル(または，根からの距離＋1）である．確率分布が与えられたとき，**重みづけパス長を最小化する**ことがこれ以後の目標である．

1例として，アクセス確率がそれぞれ $p_1 = 1/7$, $p_2 = 2/7$ および $p_3 = 4/7$ であるキー1，2，3の集合を考える．これらの3個のキーは5個の異なった探索木の形に配置することができる(**図4.36**参照)．

(4.66)にしたがって重みづけパス長を計算すると，

$$P_I^{(a)} = \frac{1}{7}(1\cdot3+2\cdot2+4\cdot1) = \frac{11}{7}$$

$$P_I^{(b)} = \frac{1}{7}(1\cdot2+2\cdot3+4\cdot1) = \frac{12}{7}$$

$$P_I^{(c)} = \frac{1}{7}(1\cdot2+2.1+4\cdot2) = \frac{12}{7}$$

$$P_I^{(d)} = \frac{1}{7}(1\cdot1+2\cdot3+4\cdot2) = \frac{15}{7}$$

$$P_I^{(e)} = \frac{1}{7}(1\cdot1+2\cdot2+4\cdot3) = \frac{17}{7}$$

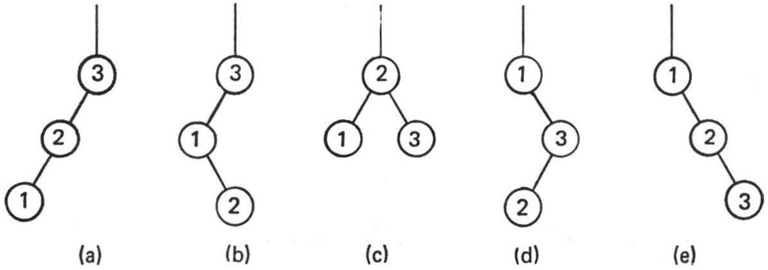


図 4.36 3個の節点からなる探索木

したがって，この例では最適な配置になるのは，完全バランス木ではなく，縮退した木(a)である．

コンパイラの走査プログラムの例について考えてみると，この問題はもう少し一般的な条件のもとで考察すべきであることがすぐにわかる：ソース文の中に現われる語は常にキー・ワードであるというわけではなく，実際には，キー・ワードであることはむしろまれである．与えられた語 k が探索木中のキーでないことを検出することは，それと隣合うより小さいキーとより大きいキーの中間に，ある外部パス長をもつ仮想的な“特別な節点”を挿入し（図 4.19 参照），これへのアクセスがあったと考えることができる．もし，探索の引数 x が2つのキー k_i と k_{i+1} の間にある確率 q_i もまた既知であると仮定すると，この情報は最適な探索木の構造を著しく変える可能性がある．したがって，探索が失敗する場合も考えてこの問題を一般化すると，重みつきパス長の全体的平均値は，次のようになる．

$$P = \sum_{i=1}^{n} p_i h_i + \sum_{j=0}^{n} q_j h'_j \qquad (4.67)$$

ここで，

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1$$

である．また，h_i は(内部)節点 i のレベルであり，h'_j は外部節点 j のレベルである．重みつきパス長の平均値は探索木の"コスト"とよぶことができるが，その理由は，それが探索に費やされる作業量の期待値の目安を表わしているからである．キー k_i の集合と確率 p_i，q_j が与えられたとき，すべての木のなかでそのコストが最小であるような構造をもつ探索木は**最適木**と呼ばれる．

最適木を見つけるためには，p，q の和が1になることを要求する必要はない．実際，これらの確率は節点へのアクセス回数を測定するという実験によって決定されるのが普通であるので，以下ではこれらの確率 p_i や q_j の代わりに，そのような回数の測定値を用いることにし，次のような記号を用いる．

a_i = 探索引数 x が k_i に等しい回数
b_j = 探索引数 x が k_j と k_{j+1} の間にある回数

便宜上，b_0 は x が k_1 よりも小さい回数を，また b_n は x が k_n より大きい回数を表わすとする(図4.37参照)．

また，平均パス長の代わりに重みつきパス長の総和を用い，これを P で表わす：

$$P=\sum_{i=1}^{n} a_i h_i + \sum_{j=0}^{n} b_j h'_j \tag{4.68}$$

このようにすることにより，測定された回数からの確率計算を行なわずに済むが，それ以外にも，最適木を見つける作業を整数だけを用いて行なえるという

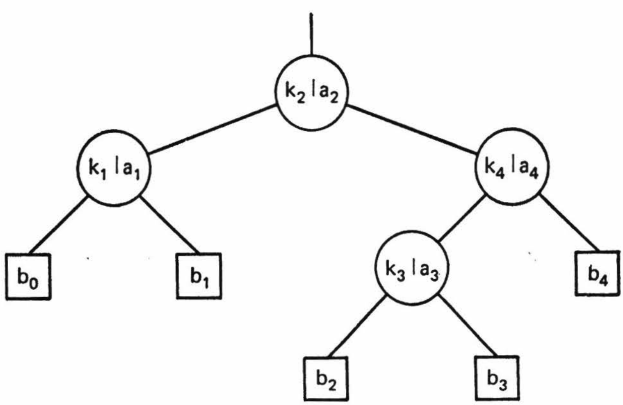


図4.37　アクセス頻度を伴なった探索木

新しい利点も生じてくる．

n 個の節点の可能な配置の数が n と共に指数関数的に増大することを考えると，n が大きいときには，最適なものを見つける作業は，むしろ絶望的であるように思える．しかしながら，最適木は，それを見つける際の助けとなるひとつの重要な性質をもっている：すなわち，そのすべての部分木もまた最適である．たとえば，もし図 4.37 の木が与えられた a, b に対して最適であるならば，キー k_3 と k_4 をもつ部分木もまた最適である．この性質により，個々の節点を最小の部分木として出発し，より大きな木を次々に見つけてゆく系統的なアルゴリズムが考えられる．木はこのようにして，"葉から根の方向"に成長してゆく．木はさかさまに描かれるのが普通であるので，これは，"ボトムアップ" (bottom-up) の方向である [4-6].

このアルゴリズムにとって基本となる方程式は (4.69) である．P をある木の重みつきパス長とし，またその根の左，右部分木のそれを P_L, P_R とする．明らかに，P は P_L と P_R および根へのアクセスの回数，すなわち，探索試行の全回数 W の和である．

$$P = P_L + W + P_R \tag{4.69}$$

$$W = \sum_{i=1}^{n} a_i + \sum_{j=0}^{n} b_j \tag{4.70}$$

W は木の**重み**とよばれる．このとき，**平均**パス長は P/W である．

これらの考察から，いくつかの隣接するキーから構成される部分木に対して，その重みとパス長を表わすための記法が必要であることがわかる．キー k_{i+1}, k_{i+2}, …, k_j をもつ節点から構成される最適な部分木 T_{ij} の重みとパス長を，w_{ij} および p_{ij} で表わすと，これらの量は再帰関係式 (4.71) と (4.72) によって定義される．

$$\begin{aligned}
w_{ii} &= b_i && (0 \le i \le n) \\
w_{ij} &= w_{i,\,j-1} + a_j + b_j && (0 \le i < j \le n) \qquad (4,71) \\
p_{ii} &= w_{ii} && (0 \le i \le n) \\
p_{ij} &= w_{ij} + \min_{i<k\le j}(p_{i,\,k_1} + p_{kj}) && (0 \le i < j \le n) \qquad (4.72)
\end{aligned}$$

この最後の方程式は，(4.69) と最適性の定義からただちに導かれる．

p_{ij} の値は約 $(1/2)n^2$ 個あり，(4.72) では $0 < j-i \le n$ 個の場合のなかからの選択が要求されるので，最小化のためには約 $(1/6)n^3$ ステップの操作が必要

である．Knuth は次のような考察から，n^2 のオーダにすることが可能であることを指摘したが，これによりこのアルゴリズムの実用性が損われずに済んでいる．

(4.72) において，最小値が得られる k の値を r_{ij} とする．r_{ij} の探索をもっと小さな区間に限定する，すなわち $j-i$ 個の評価ステップ数を減少させることが可能である．その鍵は次の事実が成立することを知ることである．すなわち，最適部分木 T_{ij} の根 r_{ij} が見つかったとするとき，その木の右側に新しい節点を追加しても，あるいは，その木の最左の節点を取除いたとしても，根を左側に移動させることはできない．これは次の関係によって表わされる．

$$r_{i,j-1} \le r_{ij} \le r_{i+1,j} \tag{4.73}$$

これにより，r_{ij} のための可能な解の探索は，範囲 $r_{i,j-1}\cdots r_{i+1,j}$ に限定され，その結果，必要となる基本ステップの総数は $O(n^2)$ になる．以上で，最適化アルゴリズムを詳細に構成する準備ができた．キーが $k_{i+1},\cdots,k_j$ の節点からなる最適木に関する次の定義を思い出そう．

1. a_i：キー k_i に対する探索の回数
2. b_i：探索の引数 x が k_i と k_{i+1} の間にある回数
3. w_{ij}：T_{ij} の重み
4. p_{ij}：T_{ij} の重みつきパス長
5. r_{ij}：T_{ij} の根のインデックス

型宣言

$$\textbf{type}\ \mathit{index} = 0\,..\,n$$

が与えられたときに，次のような配列を宣言する．

$$\begin{aligned} a &: \textbf{array}\ [1\,..\,n]\ \textbf{of}\ \mathit{integer}; \\ b &: \textbf{array}\ [\mathit{index}]\ \textbf{of}\ \mathit{integer}; \\ p, w &: \textbf{array}\ [\mathit{index}, \mathit{index}]\ \textbf{of}\ \mathit{integer}; \\ r &: \textbf{array}\ [\mathit{index}, \mathit{index}]\ \textbf{of}\ \mathit{index} \end{aligned} \tag{4.74}$$

重み w_{ij} は，a と b から単純な方法で計算されていると仮定する [(4.71) 参照]．さて，以下では w は構成すべき手続きの入力，r はその出力であると考えるが，これは r によって木構造が完全に記述されるからである．p は中間的な結果と考えられる．最も小さい部分木，すなわち，いかなる節点も含まない部分木から考え始め，より大きな木へと逐次進んでゆく．部分木 T_{ij} の幅 $j-i$ を h で

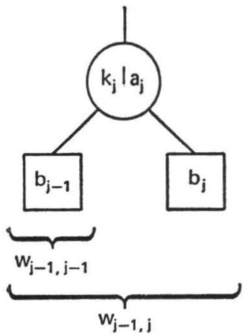


図 4.38 節点 1 個からなる最適木

表わそう．このとき，(4.72) によれば，$h=0$ のすべての木に対して，値 p_{ii} は明らかに次のように決定される．

for $i := 0$ **to** n **do** $p[i,i] := w[i,i]$ (4.75)

$h=1$ の場合には，1 個の節点からなる木を扱うが，この節点は明らかに根である (図4.38)．

```
for i := 0 to n−1 do
    begin j := i+1; p[i,j] := p[i,i]+p[j,j]; r[i,j] := j        (4.76)
    end
```

i は考えている木 T_{ij} におけるインデックスの左限を，また j は右限を表わしていることに注意せよ．$h>1$ の場合については，h が 2 から n まで変化する繰返し文を用いるが，$h=n$ の場合に完全な木 $T_{0,n}$ が張られる．各々の場合において，パス長の最小値 p_{ij} と対応する根のインデックス r_{ij} は，インデックス k が (4.73) で与えられる区間を動くような簡単な繰返し文によって決定される．

```
for h := 2 to n do
for i := 0 to n−h do
begin j := i+h;
      "find m and min = minimum(p[i,m−1]+p[m,j]) for all      (4.77)
       m such that r[i,j−1] ≤ m ≤ r[i+1,j]";
      p[i,j] := min + w[i,j];  r[i,j] := m
end
```

引用符の内部の文の詳細化は，**プログラム 4.6** 中に見つけることができる．$T_{0,n}$ の平均パス長は，いまや商 $p_{0,n}/w_{0,n}$ によって与えられ，そしてその根はインデックスが $r_{0,n}$ の節点である．

アルゴリズム (4.77) より，最適な構造を決定するための計算ステップは $O(n^2)$ であり，またそれに必要な記憶量も $O(n^2)$ であることは明らかである．しかし，これは，n が非常に大きい場合には受け入れがたい大きさである．したがって，もっと効率のよいアルゴリズムが切望される．それらのうちの 1 つに，Hu と Tucker [4-5] によって考えられたものがあるが，これは，$O(n)$ の記憶量と $O(n \cdot \log n)$ の計算ステップしか必要としないものである．しかしながら，これはキーの頻度が 0 ($a_i=0$) の場合，すなわち失敗した探索試行だけが記録されている場合，しか考えていない．$O(n)$ の記憶量と $O(n \cdot \log n)$ の計算ステップしか必要としないもう 1 つのアルゴリズムが Walker と Gotlieb [4-11] によって与えられた．このアルゴリズムは，最適木を見つける代わりに，準最適木を求めようとするものであり，**発見的**原理を利用している．基本的な考え方は次のようである．

節点 (本当の節点と特別な節点) が，そのアクセス頻度 (または確率) による重みにしたがって，直線状の物差しの上に分布していると考えよう．このとき "重心" に最も近い節点をみつけ，これを**中心節点** (centroid) とよぶ．そのインデックスは，

$$\frac{1}{w}\left(\sum_{i=1}^{n} i \cdot a_i + \sum_{j=0}^{n} j \cdot b_j\right) \tag{4.78}$$

を最も近い整数に丸めたものである．もし，すべての節点が等しい重みをもっているならば，求める最適木の根は明らかに中心節点と一致する．したがって大部分の場合，それは中心節点の近傍に存在すると推定される．ゆえに，限定された探索を行なって局所的最適値を見つけ，次に，結果として得られた 2 つの部分木にこの手続きを適用する．根が中心節点の近くに存在する度合いは，木の大きさ n とともに大きくなる．部分木が "扱い易い" 大きさになれば，先に述べた厳密なアルゴリズムを用いてその最適解を求めることができる．

4.4.10　木構造の表示

ここでは，通常のプリンタのみを使用して木構造をほどよく分りやすい図の形に**表示**するプログラムの問題を考えよう．すなわち節点としてキーを印刷し，適当な横棒および縦棒記号によりそれらを接続して木を描きたい．

ライン・プリンタでは，そのデータがテキスト・ファイル，すなわち，記号の列として表現されているので，厳密に左から右，および，上から下の順でしか進むことができない．したがって，まず最初のステップでは，木のトポロジカルな構造を厳密に反映した表現を作りあげることがよいように思われる．そして，次のステップでこの表現を通常の形式で印刷ページ上に**写像**し，節点や枝の正確な座標を計算する．

最初のステップには，木生成アルゴリズムの経験から，再帰的な解を採用するが，それはこの問題が再帰的に定義されたものだからである．プログラム 4.3 において使用したのと同様な tree と呼ばれる関数手続きを定式化する．そのパラメータ i と j は，その木に属する節点のインデックスの節囲を示している．また，その根はインデックスが r_{ij} の節点と定義される．先に進むまえに，節点を表わすための変数の型を定義しなければならない．節点には，その部分木への 2 つのポインタとキーが含まれなければならないが，次のステップで述べる目的のために，*pos* と *link* と呼ばれる 2 つの付加的なフィールドも組み込んでおく．このようにして決定された定義を (4.79) に示す．また，その結果作られた関係手続きが**プログラム 4.6** 中に与えられている．

```
type ref = ↑node;
   node = record key: alfa;
                 pos: lineposition;
                 left,right,link: ref
          end
```
(4.79)

この手続きでは，大域的なカウンタ用変数 k により，生成した節点の数を数えていることに注意せよ．k 番目の節点には k 番目のキーが割り当てられるが，キーはアルファベット順に並んでいるので，k に一定の倍率を掛けることにより各キーの**横方向** (*horizontal*) の位置が求められる．この値は他の情報とともに記憶しておく．また，これまでは便宜上キーとしては整数を用いてきたが，これ以後その型は *alfa* であると仮定する．この型は，アルファベット順序の定義された与えられた長さ (最大長) の文字配列である．

これまでに得られたものを図示すると**図 4.39** のようになる．n 個のキーの集合と計算済みの行列 r_{ij} が与えられたとき，文

$$k := 0; \quad root := tree(0, n)$$

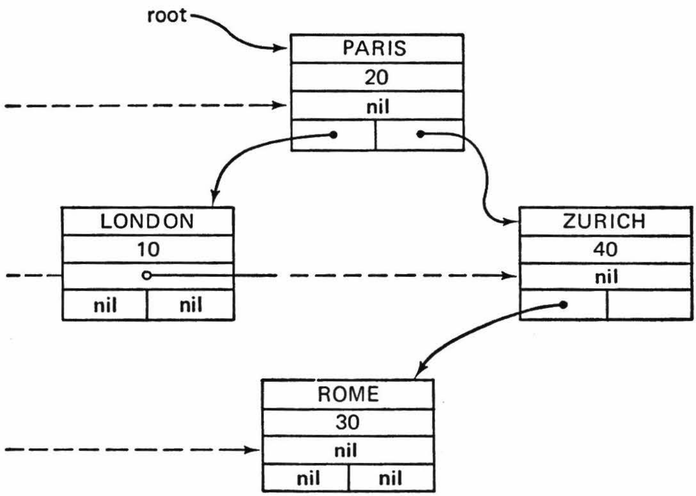


図4.39　プログラム4.6から生じた木

により，リンクされた予備的な木構造が生成されるが，これには節点の横方向の位置が記録されている．**縦方向** (*vertical*) の位置は，木の中における節点のレベルから暗黙的に決定される．

さて，次のステップ(ページ上への木の写像)に進もう．この場合，根節点から出発し，厳密に下向きに進みながら，各ステップで1行分の節点の処理を行なわなければならない．しかしながら，ある1つの行の上にある節点をどのよにしてアクセスしたらよいであろうか．この目的のために，すなわち同じ行の上の節点を連結するために，*link* と呼ばれる成分を既に導入しておいた．確立すべき結合は，**図4.39** において点線で示されている．各処理ステップでは，印刷すべき節点を連結しているリスト——これを**現在の** (*current*) リストという——は既に存在していると仮定し，また各節点の処理中にその子孫（もしあれば）を見つけ，それを連結して2番目のリスト——**次の** (*next*) リストとよぶ——を作る．1レベル下るごとに，次のリストが現在のリストとなり，次のリストには空の印がつけられる．

アルゴリズムの詳細はプログラム4.6から知ることができるが，いくつかの点を明確にするために次の注意が役立つであろう．

1. ある行の上の節点のリストは左から右に作られ，その結果，最左の節点はリストの最後になる．節点は同じ順番に訪問されなければならないので，このリストは反転しなければならないが，この反転は次のリストが現在のリストになるときに行なわれる．
2. キーを並べた印刷行——**主行** (*master line*) と呼ぶ——には横方向の枝も含まれる(**図 4.40** 参照)．変数 *u*1, *u*2, *u*3, *u*4 は，節点から左および右に出る横方向の枝の始点と終点の位置を表わしている．
3. 各主行の印刷に先だって，縦方向の枝のために 3 行分の印刷が行なわれる．

さて，プログラム 4.6 の構造について述べよう．その 2 つの主要成分は，重み分布 w が与えられたときに最適探索木を見つける手続きと，インデックスが与えられたときに木を表示する手続きである．このプログラム全体はプログラム・テキスト，特に PASCAL プログラムを処理するように作られている．最初の部分ではプログラムを読み，その識別名とキー・ワードを認識し，キー・ワード k_i および k_j と k_{j+1} の間の識別名が見つけられる回数 a_i および b_j を求める．この頻度統計を印刷したあとで，プログラムは完全バランス木のパス長の計算を行ない，ついでにその各部分木の根を決定する．その後，重みつきパス長の平均値を印刷し木を表示する．

第 3 番目の部分では，手続き *opttree* を起動し，最適探索木を計算し表示する．そして，最後に同じ手続きを用いて，キーの頻度だけを考慮した最適木を計算し表示する．

表 4.5 および**図 4.40** から**図 4.42** は，プログラム 4.6 に自分自身のプログラム・テキストを入力したときの結果を示している．この 3 つの図形の相違は，バランス木は準最適とも考えられないこと，また，キー以外の語の頻度が最適構造の決定に大きな影響を及ぼしていることを示している．

プログラム 4.6 最適探索木の発見

```
program optimaltree(input,output);
const n = 31; {no. of keys}
   kln = 10;  {max keylength}
type index = 0 .. n;
   alfa = packed array [1 .. kln] of char;
```

```
var ch: char;
      k1, k2: integer;
      id: alfa;     {identifier or key}
      buf: array [1 .. kln] of char;     {character buffer}
      key: array [1 .. n] of alfa;
      i,j,k: integer;
      a: array [1 .. n] of integer;
      b: array [index] of integer;
      p,w: array [index,index] of integer;
      r: array [index,index] of index;
      suma, sumb: integer;
function baltree(i,j: index): integer;
   var k: integer;
begin k := (i+j+1) div 2; r[i,j] := k;
   if i ≥ j then baltree := b[k] else
      baltree := baltree(i,k-1) + baltree(k,j) + w[i,j]
end {baltree} ;
procedure opttree;
   var x, min: integer;
      i,j,k,h,m: index;
begin {argument: w, result: p,r}
   for i := 0 to n do p[i,i] := w[i,i];  {width of tree h = 0}
   for i := 0 to n-1 do              {width of tree h = 1}
   begin j := i+1;
      p[i,j] := p[i,i] + p[j,j];  r[i,j] := j
   end ;
   for h := 2 to n do                  { h = width of considered tree }
   for i := 0 to n-h do                { i = left index of considered tree }
   begin j := i+h;                     { j = right index of considered tree }
      m := r[i,j-1];  min := p[i,m-1] + p[m,j];
      for k := m+1 to r[i+1,j] do
      begin x := p[i,k-1] + p[k,j];
         if x < min then
            begin m := k;  min := x
            end
      end ;
      p[i,j] := min + w[i,j];  r[i,j] := m
   end
end {opttree} ;
```

```
procedure printtree;
   const lw = 120;      {line width of printer}
   type ref = ↑node;
      lineposition = 0 .. lw;
      node = record key: alfa;
                    pos: lineposition;
                    left, right, link: ref
             end ;
   var root, current, next: ref;
      q,q1,q2: ref;
      i, k: integer;
      u, u1, u2, u3, u4: lineposition;
   function tree(i,j: index): ref;
      var p: ref;
   begin if i = j then p := nil else
      begin new(p);
         p↑.left := tree(i, r[i,j]−1);
         p↑.pos := trunc((lw−kln)*k/(n−1)) + (kln div 2); k := k+1;
         p↑.key := key[r[i,j]];
         p↑.right := tree(r[i,j], j)
      end ;
      tree := p
   end ;
begin k := 0;  root := tree(0,n);
   current := root; root↑.link := nil;
   next := nil;
   while current ≠ nil do
   begin {proceed down; first write vertical lines}
      for i := 1 to 3 do
      begin u := 0; q := current;
         repeat u1 := q↑.pos;
            repeat write(' '); u := u+1
            until u = u1;
            write('|'); u := u+1; q := q↑.link
         until q = nil;
         writeln
      end ;
      {now print master line; descending from nodes on current list
       collect their descendants and form next list}
```

```
    q := current; u := 0;
    repeat unpack(q↑.key, buf, 1);
        {center key about pos}  i := kln;
        while buf[i] = ' ' do  i := i-1;
        u2 := q↑.pos - ((i-1) div 2); u3 := u2+i;
        q1 := q↑.left; q2 := q↑.right;
        if q1 = nil then u1 := u2 else
            begin u1 := q1↑.pos; q1↑.link := next; next := q1
            end ;
        if q2 = nil then u4 := u3 else
            begin u4 := q2↑.pos+1; q2↑.link := next; next := q2
            end ;
        i := 0;
        while u < u1 do begin write(' '); u := u+1 end ;
        while u < u2 do begin write('-'); u := u+1 end ;
        while u < u3 do begin i := i+1; write(buf[i]); u := u+1 end ;
        while u < u4 do begin write('-'); u := u+1 end ;
        q := q↑.link
    until q = nil;
    writeln;
    {now invert next list and make it current list}
    current := nil;
    while next ≠ nil do
        begin q := next; next := q↑.link;
            q↑.link := current; current := q
        end
  end
end {printtree} ;

begin {initialize table of keys and counters}
  key[ 1] := 'ARRAY';     key[ 2] := 'BEGIN';
  key[ 3] := 'CASE';      key[ 4] := 'CONST';
  key[ 5] := 'DIV';       key[ 6] := 'DOWNTO';
  key[ 7] := 'DO';        key[ 8] := 'ELSE';
  key[ 9] := 'END';       key[10] := 'FILE';
  key[11] := 'FOR';       key[12] := 'FUNCTION';
  key[13] := 'GOTO';      key[14] := 'IF';
  key[15] := 'IN';        key[16] := 'LABEL';
  key[17] := 'MOD';       key[18] := 'NIL';
  key[19] := 'OF';        key[20] := 'PROCEDURE';
```

```
key[21] := 'PROGRAM'; key[22] := 'RECORD';
key[23] := 'REPEAT';  key[24] := 'SET';
key[25] := 'THEN';    key[26] := 'TO';
key[27] := 'TYPE';    key[28] := 'UNTIL';
key[29] := 'VAR';     key[30] := 'WHILE';
key[31] := 'WITH';
for i := 1 to n do
   begin a[i] := 0;  b[i] := 0
   end ;
b[0] := 0; k2 := kln;
{scan input text and determine a and b}
while ¬eof(input) do
begin read(ch);
   if ch in ['A' .. 'Z'] then
   begin {identifier or key} k1 := 0;
      repeat if k1 < kln then
             begin k1 := k1+1; buf[k1] := ch
             end ;
         read(ch)
      until ¬ch in ['A' .. 'Z', '0' .. '9']);
      if k1 ≥ k2 then k2 := k1 else
      repeat buf[k2] := ' '; k2 := k2−1
      until k2 = k1;
      pack(buf,1,id);
      i := 1; j := n;
      repeat k := (i+j) div 2;
         if key[k] ≤ id then i := k+1;
         if key[k] ≥ id then j := k−1;
      until i > j;
      if key[k] = id then a[k] := a[k] + 1 else
         begin k := (i+j) div 2; b[k] := b[k]+1
         end
   end else
   if ch = '''' then
      repeat read(ch) until ch = '''' else
   if ch = '{' then
      repeat read(ch) until ch = '}'
end ;
```

```
  writeln ('KEYS AND FREQUENCIES OF OCCURRENCE:');
  suma := 0; sumb := b[0];
  for i := 1 to n do
  begin suma := suma+a[i]; sumb := sumb+b[i];
     writeln(b[i-1], a[i], '   ', key[i])
  end ;
  writeln(b[n]);
  writeln('        ------        ------');
  writeln(sumb, suma);
  {compute w from a and b}
  for i := 0 to n do
  begin w[i,i] := b[i];
     for j := i+1 to n do w[i,j] := w[i,j-1] + a[j] + b[j]
  end ;
  write('AVERAGE PATH LENGTH OF BALANCED TREE= ');
  writeln(baltree(0,n)/w[0,n]:6:3); printtree;
  opttree;
  write('AVERAGE PATH LENGTH OF OPTIMAL TREE= ');
  writeln(p[0,n]/w[0,n]:6:3);  printtree;
  {now consider keys only, setting b = 0}
  for i := 0 to n do
  begin w[i,i] := 0;
     for j := i+1 to n do w[i,j] := w[i,j-1] + a[j]
  end ;
  opttree;
  writeln('OPTIMAL TREE CONSIDERING KEYS ONLY');
  printtree
end
```

表4.5 キーと出現頻度

4	7	ARRAY
14	27	BEGIN
19	0	CASE
15	2	CONST
8	5	DIV
0	0	DOWNTO
0	20	DO
0	8	ELSE
0	28	END
1	0	FILE
0	12	FOR
0	2	FUNCTION
0	0	GOTO
9	13	IF
23	2	IN
208	0	LABEL
22	0	MOD
17	10	NIL
24	7	OF
17	2	PROCEDURE
0	1	PROGRAM
53	1	RECORD
6	8	REPEAT
16	0	SET
10	13	THEN
0	12	TO
6	2	TYPE
1	8	UNTIL
39	5	VAR
0	8	WHILE
0	0	WITH
37		
549	203	

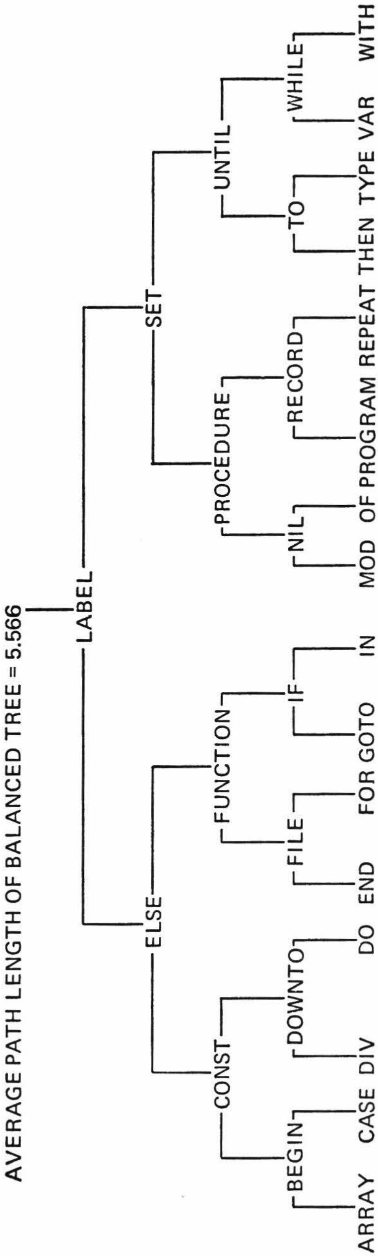


図 4.40　完全バランス木

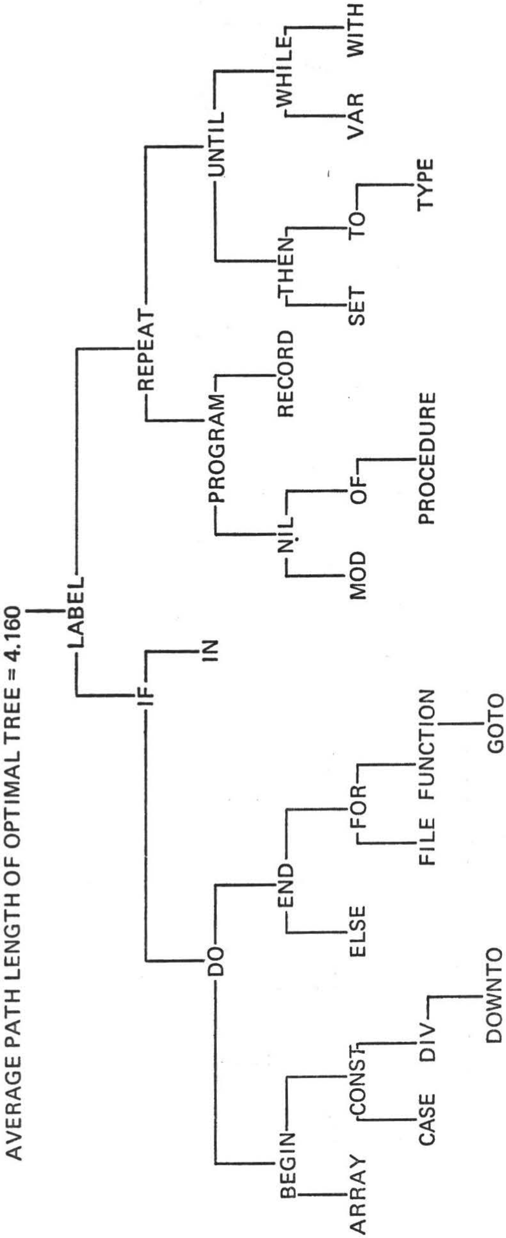

AVERAGE PATH LENGTH OF OPTIMAL TREE = 4.160
LABEL
IF
IN
REPEAT
DO
PROGRAM
UNTIL
BEGIN
END
NIL
RECORD
THEN
WHILE
ARRAY
CONST
ELSE
FOR
MOD
OF
SET
TO
VAR
WITH
CASE
DIV
FILE
FUNCTION
PROCEDURE
TYPE
DOWNTO
GOTO

図 4.41 最適探索木

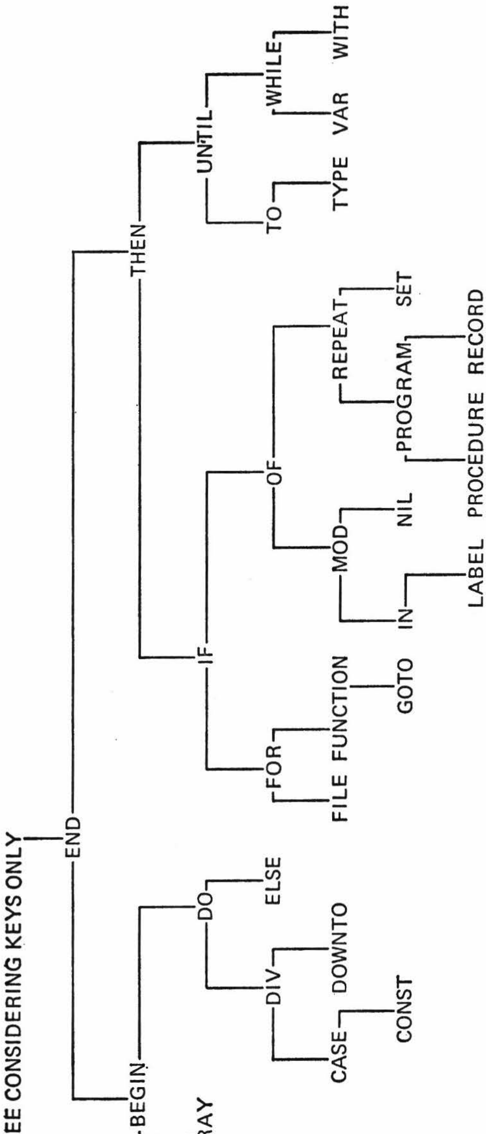


図 4.42　キーだけを考慮した最適木

4.5 多 分 木

これまでは，各節点が高々2つの子孫しかもたない木すなわち2分木についてだけ議論してきた．これは，たとえば親族関係を“家系の観点”，すなわち各人にその両親を対応させるという観点から表現したいと思うときには，完全に満足できるものである．なぜなら，誰も2人以上の両親を持つことはないからである．しかし，もし“子孫の観点”から表現したい場合にはどうであろうか．2人以上の子供のいる人を表現するために，その木は多数の枝をもつ節点を含むであろう．適当な言葉がないので，われわれはそのような木を**多分木**とよぶことにする．

もちろん，そのような構造が何か特別に変っているということはなく，われわれはこれまでに，そのような状況に対処するためのプログラミングおよびデータ定義上のすべての機能を学んできた．たとえば，もし子供の数に絶対的な上限があれば（これは確かにいくらか未来的な仮定ではあるが），子供を表わすレコードを成分とする配列によって人を表現することができる．しかしながら，もし子供の数が人ごとに大きく異なるならば，この方法では記憶装置の利用率が悪くなってしまうであろう．この場合には，**子供達** (offspring) を線形リストの形に連結し，最も年少（または最も年長）の子供へのポインタを親に割り当てる方が適切であろう．(4.80)はこの場合の型定義の一例であり，また**図 4.43** にはそのデータ構造の例が示されている．

```
type person = record name: alfa;
                     sibling: ↑person;
                     offspring: ↑person
              end                          (4.80)
```

この図を，45° 傾けてみると，完全な2分木のように見えることに気がつくが，この見方は誤解を招きやすい．なぜなら，2つのポインタは機能的には全く異なった意味を持っているからである．**兄弟** (sibling) を子供として扱っておきながら罰を受けずに逃げるようなことはしないのが普通であり，それゆえデータの定義を構成する場合でも，そのようにすべきではない．各人のレコードにもっと多くの成分を導入することにより，この例をより複雑なデータ構造に拡張することは容易であり，したがって，もっと複雑な親族関係を表現することが可能である．子供と兄弟の関係からは一般に導くことが不可能な関係としては，夫婦の関係や父母に関する逆の関係などがある．そのような構造は複雑

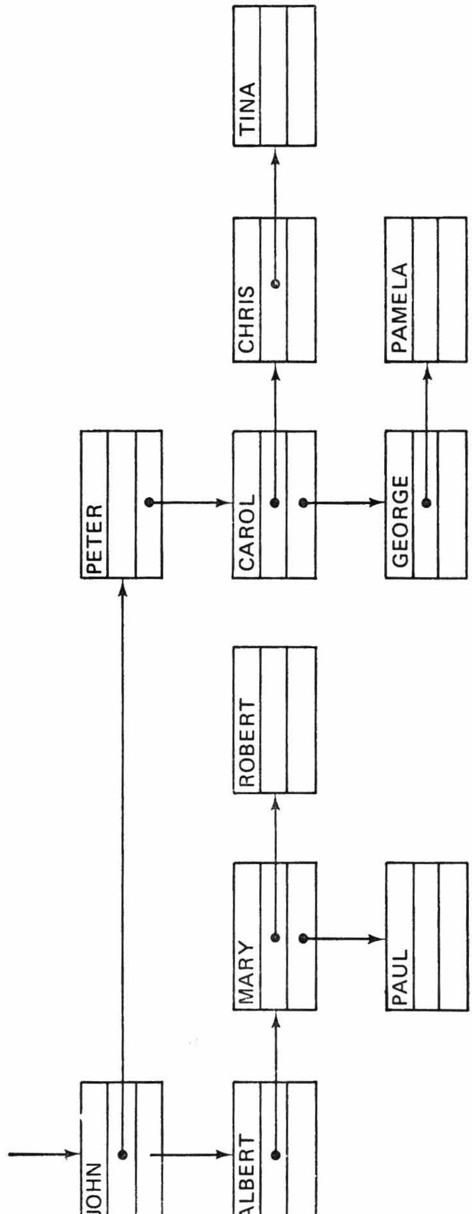

JOHN
ALBERT
MARY
PAUL
ROBERT
PETER
CAROL
GEORGE
PAMELA
CHRIS
TINA

図 4.43　多分木の例

な“関係データ・バンク”に急速に成長し，いくつかの木をその中に写像することが可能になるであろう．そのような構造を操作するアルゴリズムは，データ定義と密接に関連しており，したがって一般的な規則や広く適用可能な手法を指定することは無意味である．

しかしながら，多分木の非常に実際的な応用分野でありかつ一般的にも興味のあるものがある．これは，挿入や削除が必要ではあるが計算機の主記憶ではその容量が十分でないばかりでなく，長期の記憶のために使用するにはそのコストがかかりすぎるような大規模探索木の構成と保守の問題である．

この場合，木の節点は，ディスク記憶装置のような2次記憶媒体に記憶されると仮定しよう．本章で導入する動的データ構造は，2次記憶媒体に組み込む場合には特に適したものである．基本的な考え方は，主記憶の番地の代わりに，ディスク記憶装置の番地を使用することである．しかし，100万項目のデータ集合のために2分木を使うとすると，平均約 $\log_2 10^6 \cong 20$ の探索ステップを必要とし，かつ，各ステップには1回の(固有の回転待ち時間を伴なう)ディスク・アクセスが含まれる．したがって，もっとアクセス回数の少ない記憶組織法が強く望まれる．多分木はこの問題に対する1つの完全な解である．2次記憶上のある項目がアクセスされる場合，ある一群の項目全体もまた余分なコストを必要とせずにアクセスすることができる．これは，木を部分木に分割し，その部分木を一度にアクセスされる単位として表現する方法を示唆している．そのような部分木を**ページ**と呼ぶことにする．**図4.44**は，7個の節点からなるページに分割された木を示している．

各ページは，1回のディスク・アクセスによってアクセスされるので，ディスク・アクセス回数を著しく節約することができる．たとえば，各ページに

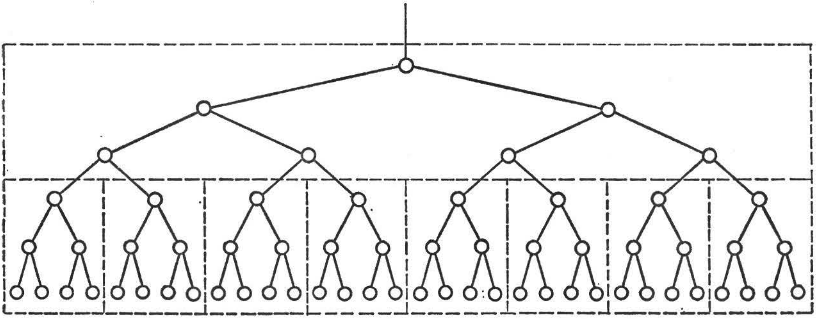


図4.44 “ページ”に分割された木

100 個の節点を置くとすると（この数字は穏当なものである），100 万項目からなる探索木は，20 ではなく平均 $\log_{100} 10^6 = 3$ 回のページ・アクセスによって探索可能であろう．しかしながら，もちろん木が“ランダム”に成長した場合には，最悪の場合 10^4 回のページ・アクセスが必要である！　多分木の場合，制御された成長方式が必須なことはほとんど明らかである．

4.5.1　B　木

制御された成長の基準を考える場合，完全なバランスを要求するものは，バランス化のためのオーバ・ヘッドが大きすぎるので，ただちにその対象から外されることになる．したがって，いくらか緩やかな基準を採用すべきである．非常に賢明な基準が R. Bayer [4-2] により 1970 年に提案された．これは，与えられた n に対して，（1 つを除いて）すべてのページが n から $2n$ までの節点を含むというものである．したがって，木が N 個の項目からなり，最大ページ・サイズが $2n$ 節点である場合，最悪な場合でも必要なページ・アクセス回数は $\log_n N$ である．――むろん，ページ・アクセスが全探索時間中最も大きなウエイトを占めることは明らかである．さらに，重要な記憶利用率は最低でも 50% である．これは，少なくともページの半分がいつも満たされているからである．このような利点に加えて，この方法は探索，挿入および削除のためのアルゴリズムが比較的簡単である．以下ではこれらについて詳細に検討する．

基本となるデータ構造は **B 木** (B-tree) と呼ばれ，次のような特徴をもっている：n は B 木の位数といわれる．

1. 各ページは，高々 $2n$ 個の項目（キー）しか含まない．
2. 根のページを除いて，各ページは少なくとも n 個の項目を含む．
3. 各ページは，葉のページであり子孫を持たないか，あるいは $m+1$ 個の子孫をもつ．ただし m はそのページに含まれるキーの数である．
4. 葉のページは，すべて同一レベルに現われる．

図 4.45 は位数 2，レベル 3 の B 木を示している．すべてのページには，2, 3 または 4 個の項目が含まれるが，根は例外であり 1 個の項目しか含まないことが許される．すべての葉はレベル 3 に現われる．先祖のページのキーの間に子孫のページを挿入して，B 木を 1 レベルに押しつぶすと，キーは左から右に昇順になっている．この構造は 2 分探索木の自然な拡張であり，与えられたキーを持つ項目の探索法を定めている．**図 4.46** に示される形のページおよび与えられ

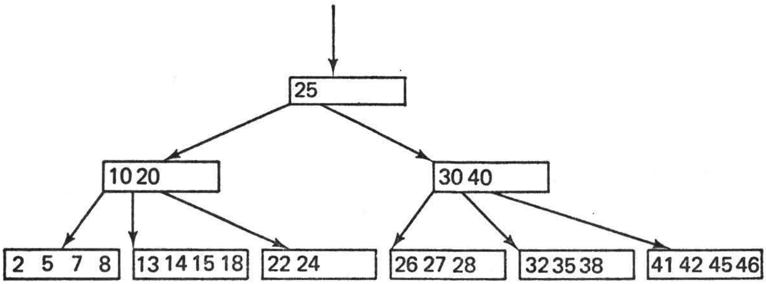


図 4.45 位数 2 の B木

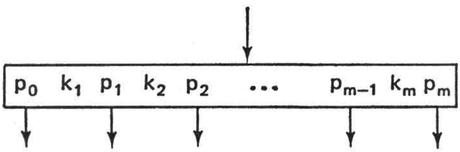


図 4.46 m 個のキーをもつ B木のページ

た探索引数 x について考えよう．このページが主記憶に移されたと仮定すると，キー k_1, …, k_m 中の探索は通常の方法によって行なうことができる．もし，m が十分に大きいときには，たぶん 2 分探索法が使われるであろう．また m が小さいときには，通常の順次的探索で十分である（主記憶中での探索に要する時間は，2 次記憶から主記憶にページを移すための時間に比べれば無視しうることに注意せよ）．もし探索が失敗したら，次の状況のいずれかである．

1. $k_i < x < k_{i+1}$ $(1 \leq i < m)$．ページ $p_i\uparrow$ について探索を続ける．
2. $k_m < x$．ページ $p_m\uparrow$ について探索を続ける．
3. $x < k_1$．ページ $p_0\uparrow$ について探索を続ける．

もし，指示されたポインタが **nil**，すなわち子孫のページが存在しないならば，木の中にはキー x をもつ項目は存在せず，探索は停止される．

驚いたことに，B木への挿入も比較的簡単である．$m(< 2n)$ 個の項目しか含まれないページに挿入するときには，その挿入の過程はこのページ内に限定される．木構造に影響を及ぼしたりあるいは新しいページが割り付けられる可能性のあるのは，既にいっぱいのページに挿入が行なわれる場合だけである．この場合に何が起こるかを理解するために図 4.47 を参照するが，これは位数

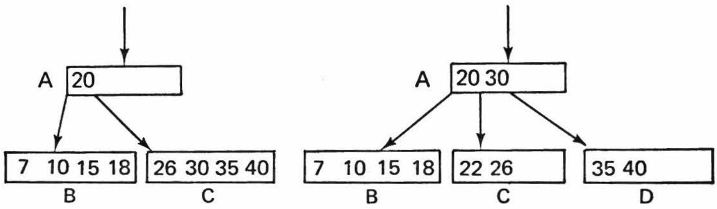


図4.47　B木へのキー22の挿入

2の B 木へのキー 22 の挿入を描いたものである．挿入は次のようなステップで行なわれる．

1. キー 22 が存在しないことがわかる：ページ C は既にいっぱいであるので，ページ C への挿入は不可能である．
2. ページ C を 2 つのページに分割する（すなわち新しいページ D を割り付ける）．
3. $m+1$ 個のキーを C と D 上に等しく分配し，中間のキーを 1 レベル上に動かしその先祖 A に移す．

この大変エレガントな方式は，B木の特徴的性質すべてを保存する．特に，分割ページは正確に n 個の項目を含んでいる．もちろん先祖のページへの挿入により，そのページが再びあふれ，したがって**分割は伝播してゆく**かも知れない．極端な場合には，根まで伝播してゆく可能性があるが，これはB木がその高さを増加させる唯一の仕方である．B木は，このように奇妙な成長の仕方をする：それは葉から根に向かって成長する．

次に，これらの概略的記述から詳細なプログラムを作りあげよう．分割過程が探索パスをさかのぼって伝播してゆくことから，再帰的定式化が最も便利であることは既に明らかである．したがってプログラムの構造は，その詳細は別として，概略はバランス木への挿入の場合と似ているであろう．

まず初めに，ページ構造の定義を定式化しなければならない．項目の集合は配列の形で定式化する．

```
type page = record m: index;
                   p0: ref;
                   e: array[1 .. nn] of item
            end                                  (4.81)
```

ここで

$$\textbf{const}\ nn = 2*n;$$
$$\textbf{type}\ ref = \uparrow page;$$
$$index = 0\,..\,nn$$

かつ

$$\textbf{type}\ item = \textbf{record}\ key\colon integer;\quad p\colon ref;\quad count\colon integer\ \textbf{end} \tag{4.82}$$

項目の成分 *count* は，項目に付随した他のすべての情報を代表するものとするが，これは実際の探索の過程では何の役割ももたない．各ページには，$2n$ 個の項目のための場所が用意されていることに注意せよ．成分 m は，そのうち何個の場所が実際に使われているかを示している．$m \geq n$ であるので（根ページは除く），最低 50% の記憶利用率が保証される．

B木の探索と挿入のアルゴリズムは，**プログラム 4.7** の一部分であり，*search* と呼ばれる手続きとして定式化されている．その主要な構造は単純であり，簡単な2分木の探索を思い出させるものであるが，分岐の決定が二者択一でないところが異なっている．その代わり，“ページ内探索”は配列 e に関する2分探索によって表現されている．

挿入アルゴリズムが別の手続きとして定式化されているのは，単に見やすさのためだけである．それは，ある項目を木の上側に(根の方向に)移動させることを *search* が指示した後で起動される．この事実は論理型の出力パラメータ h によって示される：h はバランス木への挿入アルゴリズムの場合と同様な役割をもっている．その場合には，部分木が成長したことを h は示していた．今の場合 h が *true* ならば，2番目の出力パラメータ u は移動される項目を表わす．挿入は仮想的なページすなわち図 4.19 の“特別な節点”から始められる：この新しい項目は，本当に挿入される葉ページにパラメータ u としてただちに渡される．この方式の概略が(4.83)に述べられている．

主プログラム中における *search* の呼出し後に，パラメータ h が *true* になったならば，これは根ページの分割を指示している．根ページには特別な役目があるので，この過程は別個にプログラムしなければならない．そこでは，新らしい(根)ページの割付けと，パラメータ u によって与えられる1つの項目

```
procedure search(x: integer; a: ref; var h: boolean; var u: item);
begin if a = nil then
      begin {x is not in tree}
         Assign x to item u, set h to true, indicating that an
         item u is passed up in the tree
      end else
       with a↑ do
       begin {search x on page a↑}
          binary array search;
          if found then
            increment the relevant item's occurrence count else
          begin search(x, descendant, h, u);
              if h then {an item u is being passed up}
                  if (no. items on a↑) < 2n then
                    insert u on page a↑ and set h to false
                  else split page and pass middle item up
          end
       end
end                                                          (4.83)
```

の挿入だけが行なわれる．その結果，新しい根ページには１つの項目だけが含まれることになる．その詳細はプログラム 4.7 から知ることができる．

次の一連のキーの挿入により，プログラム 4.7 から構成されたＢ木が図 4.48 に示されている．

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46
27 8 32; 38 24 45 25;

セミコロンは，各ページ割付けの際の"スナップショット"の位置を示している．最後のキーの挿入により，２回の分割が起こり，２つの新しいページが割り付けられる．

このプログラムにおける **with** 句の特別な重要性に注意してほしい．これはその枠組 (4.83) においても既に明らかである．まず第一に，**with** 句は，それが前置された文の中では，ページ成分の識別名は自動的にページ $a\uparrow$ を参照することを示している．実際，もしページが２次記憶に割り付けられている場合には――大規模なデータ・バンク・システムでは必然的にそうなっているに違いない――**with** 句はさらに，指定されたページの主記憶への転送を意味する

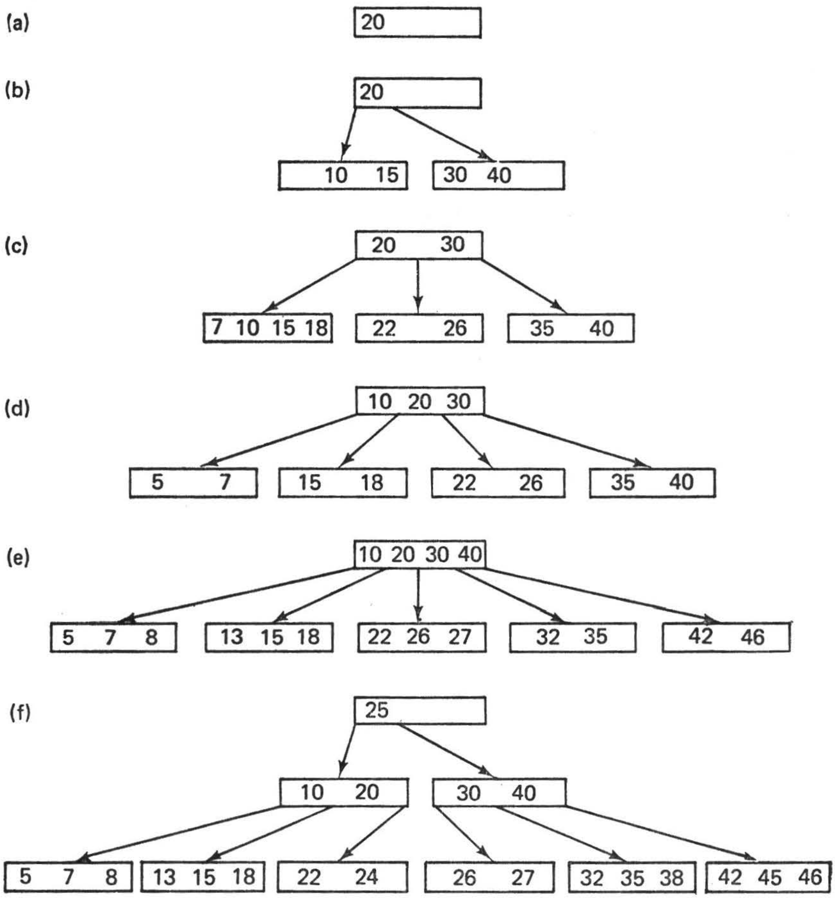


図 4.48 位数 2 の B木の成長

と解釈することができる．したがって，*search* が起動されるたびに 1 ページが主記憶に割り付けられ，また最大 $k = \log_n N$ 回の再帰呼出しが必要なので，もし木が N 個の項目を含むならば，**主記憶に k ページ**を収容できなければならない．これは，ページ・サイズ $2n$ を限定する 1 つの要因である．また，根ページは主記憶に常駐させておくのが最もよいが，これは，どのような質問も必ず根ページから開始されるからである．

B木による記憶組織法のもう 1 つの利点は，データ・バンク全体を純粋に順次的に更新するのに適しており，またそれを経済的に行なえることである．各

ページは主記憶にただ1回しか転送されない.

B木からの項目の削除は，原理的にはかなり単純であるが，その詳細はこみ入っている．2つの異なった状況を区別しなければならない：

1. 削除される項目が葉ページにある：この場合の削除アルゴリズムは単純明解である.
2. その項目が葉ページにない：それと隣合う2つの項目のいずれかで置き換えるが，これらの項目は葉ページにあり，したがって容易に削除できる.

場合2において，隣合うキーを見つけることは，2分木からの削除の場合と同様である．葉ページPに向かって，最右のポインタに沿っておりてゆき，P中の最右の項目で削除すべき項目を置き換え，しかる後Pのサイズを1だけ減らす.

いかなる場合でも，サイズを減らした後では，減らされたページ上の項目数 m をチェックしなければならない．なぜなら，もし $m<n$ ならば，B木の基本的特徴に違反するからである．このときには，何らかの付加的な動作を行なわなければならない：この**アンダー・フロー**条件は，論理型の変数パラメータ h によって示される.

これを救済する唯一の方法は，隣りのページの1つから1つの項目を借りて来る，あるいは取ってくることである．これには，ページQを主記憶に転送するという比較的高価な操作が含まれるので，この望ましからざる状況をできるだけ有効に利用し，一度に1個以上の項目を取ってくることを考える．P, Q上に項目を均等に分配するのが通常のやり方である．これは**バランス化**と呼ばれる.

もちろんQが既にその最小サイズ n になっており，取ってくるべき項目がないことが起り得る．この場合には，ページPおよびQ上の項目の総数は $2n-1$ である．したがって，これらのページを1つのページに統合し，PとQの先祖(親)のページからその中間の項目を加え，ページQを廃棄すればよい．これは，ちょうどページ分割の逆の過程である．この過程は，図4.47において，キー22が削除されたと考えれば理解しやすい.

先祖のページの中間項目の削除により，そのサイズが再び許容限界 n 以下になり，その結果特別な動作（バランス化または統合）をさらに次のレベルで行なうことが必要になるかもしれない．極端な場合には，ページ統合が根まで伝播するであろう．もし，根のサイズが0に減少したならばそれは除去され,

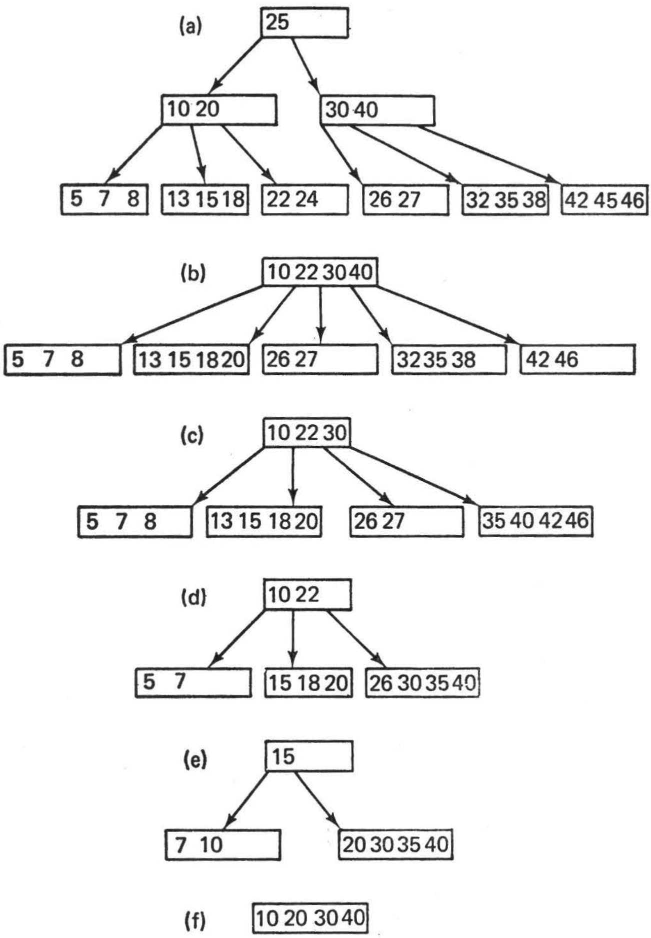


図 4.49 位数 2 の B-木の崩壊

したがって B 木の高さが減少する．実際，これは B 木の高さが縮む唯一の方法である．

図 4.49 は，キー

25　45　24; 38　32; 8　27　46　13　42; 5　22　18　26; 7　35　15;

の一連の削除により，図 4.48 の B 木が次第に崩壊してゆく様子を示したものである．セミコロンは，再び“スナップショット”のとられる位置，すなわちペ

ージが廃棄される位置を示している．この削除アルゴリズムは，プログラム4.7中に手続きとして含まれている．その構造がバランス木からの削除アルゴリズムに似ていることは特に注目に値する．

プログラム 4.7　B木の探索，挿入および削除

```
program Btree(input,output);
{B-tree search, insertion and deletion}
const n = 2; nn = 4;  {page size}
type ref = ↑page;
   item = record key: integer;
                 p: ref;
                 count: integer;
          end ;
   page = record m: 0 .. nn;  {no. of items}
                 p0: ref;
                 e: array [1 .. nn] of item;
          end ;
var root, q: ref; x: integer;
   h: boolean;  u: item;

procedure search(x: integer; a: ref; var h: boolean; var v: item);
{Search key x on B-tree with root a; if found, increment counter.
 otherwise insert an item with key x and count 1 in tree. If an item
 emerges to be passed to a lower level, then assign it to v;
 h := "tree a has become higher"}
   var k,l,r: integer; q: ref; u: item;
   procedure insert;
      var i: integer; b: ref;
   begin {insert u to the right of a↑.e[r]}
      with a↑ do
      begin if m < nn then
         begin m := m+1; h := false;
            for i := m downto r+2 do e[i] := e[i−1];
            e[r+1] := u
         end else
         begin {page a↑ is full; split it and assign the emerging
                item to v}     new(b);
            if r ≤ n then
            begin if r = n then v := u else
```

```
              begin v := e[n];
                 for i := n downto r+2 do e[i] := e[i-1];
                 e[r+1] := u
              end ;
           for i := 1 to n do b↑.e[i] := a↑.e[i+n]
         end else
         begin {insert u in right page} r := r-n; v := e[n+1]
           for i := 1 to r-1 do b↑.e[i] := a↑.e[i+n+1];
           b↑.e[r] := u;
           for i := r+1 to n do b↑.e[i] := a↑.e[i+n]
         end ;
         m := n; b↑.m := n; b↑.p0 := v .p; v .p := b
       end
     end {with}
   end {insert} ;
begin {search key x on page a↑; h = false}
   if a = nil then
   begin {item with key x is not in tree} h := true;
     with v do
     begin key := x; count := 1; p := nil
     end
   end else
   with a↑ do
   begin l := 1; r := m; {binary array search}
     repeat k := (l+r) div 2;
       if x ≤ e[k] .key then r := k-1;
       if x ≥ e[k] .key then l := k+1;
     until r < l;
     if l-r > 1 then
     begin {found} e[k] .count := e[k] .count + 1; h := false
     end else
     begin {item is not on this page}
       if r = 0 then q := p0 else q := e[r] .p;
       search(x,q,h,u); if h then insert
     end
   end
end {search} ;
```

```
procedure delete(x: integer; a: ref; var h: boolean);
{search and delete key x in b-tree a; if a page underflow is
 necessary, balance with adjacent page if possible, otherwise merge;
 h := "page a is undersize"}
   var i,k,l,r: integer; q: ref;
   procedure underflow(c,a: ref; s: integer; var h: boolean);
      {a = underflow page, c = ancestor page}
      var b: ref; i,k,mb,mc: integer;
   begin mc := c↑.m;  {h = true, a↑.m = n-1}
      if s < mc then
      begin {b := page to the right of a}  s := s+1;
         b := c↑.e[s].p; mb := b↑.m; k := (mb-n+1) div 2;
         {k = no. of items available on adjacent page b}
         a↑.e[n] := c↑.e[s]; a↑.e[n] .p := b↑.p0;
         if k > 0 then
         begin {move k items from b to a}
            for i := 1 to k-1 do a↑.e[i+n] := b↑.e[i];
            c↑.e[s] := b↑.e[k]; c↑.e[s] .p := b;
            b↑.p0 := b↑.e[k] .p; mb := mb-k;
            for i := 1 to mb do b↑.e[i] := b↑.e[i+k];
            b↑.m := mb; a↑.m := n-1+k; h := false
         end else
         begin {merge pages a and b}
            for i := 1 to n do a↑.e[i+n] := b↑.e[i];
            for i := s to mc-1 do c↑.e[i] := c↑.e[i+1];
            a↑.m := nn; c↑.m := mc-1; {dispose(b)}
         end
      end else
      begin {b := page to the left of a}
         if s = 1 then b := c↑.p0 else b := c↑.e[s-1] .p;
         mb := b↑.m + 1; k := (mb-n) div 2;
         if k > 0 then
         begin {move k items from page b to a}
            for i := n-1 downto 1 do a↑.e[i+k] := a↑.e[i];
            a↑.e[k] := c↑.e[s]; a↑.e[k] .p := a↑.p0; mb := mb-k;
            for i := k-1 downto 1 do a↑.e [i]:= b↑.e[i+mb];
            a↑.p0 := b↑.e[mb] .p;
            c↑.e[s] := b↑.e[mb]; c↑.e[s] .p := a;
            b↑.m := mb-1; a↑.m := n-1+k; h := false
         end else
```

```
        begin {merge pages a and b}
           b↑.e[mb] := c↑.e[s]; b↑.e[mb] .p := a↑.p0;
           for i := 1 to n-1 do b↑.e[i+mb] := a↑.e[i];
           b↑.m := nn; c↑.m := mc-1; {dispose(a)}
        end
     end
  end {underflow} ;

  procedure del(p: ref; var h: boolean);
     var q: ref;     {global a,k}
  begin
     with p↑ do
     begin q := e[m] .p;
        if q ≠ nil then
        begin del(q,h); if h then underflow(p,q,m,h)
        end else
        begin p↑.e[m] .p := a↑.e[k] .p; a↑.e[k] := p↑.e[m];
           m := m-1; h := m<n
        end
     end
  end {del} ;

begin {delete}
  if a = nil then
  begin writeln ('KEY IS NOT IN TREE'); h := false
  end else
  with a↑ do
  begin l := 1; r := m;  {binary array search}
     repeat k := (l+r) div 2;
        if x ≤ e[k] .key then r := k-1;
        if x ≥ e[k] .key then l := k+1;
     until l > r;
     if r=0 then q := p0 else q := e[r] .p;
     if l-r > 1 then
     begin {found, now delete e[k]}
        if q = nil then
        begin {a is a terminal page}  m := m-1; h := m<n;
           for i := k to m do e[i] := e[i+1];
        end else
```

```
            begin del(q,h); if h then underflow(a,q,r,h)
            end
        end else
        begin delete(x,q,h); if h then underflow(a,q,r,h)
        end
    end
end {delete} ;

procedure printtree(p: ref; l: integer);
    var i: integer;
begin if p ≠ nil then
    with p↑ do
    begin for i := 1 to l do write('      ');
          for i := 1 to m do write(e[i].key: 4);
           writeln;
           printtree(p0,l+1);
           for i := 1 to m do printtree(e[i] .p, l+1)
    end
end ;

begin root := nil; read(x);
    while x ≠ 0 do
    begin writeln('SEARCH KEY', x);
        search(x,root,h,u);
        if h then
        begin {insert new base page} q := root; new(root);
            with root↑ do
                begin m := 1; p0 := q; e[1] := u
                end
        end ;
        printtree(root,1); read(x)
    end ;
    read(x);
    while x ≠ 0 do
    begin writeln('DELETE KEY', x);
        delete(x,root,h);
        if h then
        begin {base page size was reduced}
```

```
            if root↑.m = 0 then
            begin q := root; root := q↑.p0; {dispose(q)}
            end
        end ;
        printtree(root,1); read(x)
    end
end
```

B 木の性能に関する詳細な解析が行なわれ，その結果が参考文献 (Bayer, McCreight) に報告されている．特に，最適なページ・サイズ n に関する問題が扱われているが，これは利用可能な記憶装置および計算機システムの特性に強く依存する．

B 木方式のいくつかの変形が Knuth [2-7] 第 3 巻 pp. 476～479 に論ぜられている．ひとつの注目すべき意見は，ページ統合と同様にページ分割も遅らせるべきであり，初めは隣りのページとのバランス化を試みるべきであるというものである．これは別として，提案されている改良は末梢的な効果しかもたらさないように思われる．

4.5.2 2分B木

B 木のうちで最も興味がないと思われるのは，位数 1 (n=1) の B 木である．しかし，このような場合に対しても注意を払うことが価値のあることもある．もちろん位数 1 のB 木は，2 次記憶に貯えられるような大規模で，順序のある，インデックスつきデータ集合を表現するには有用ではない．なぜなら，全ページのうち約 50% は 1 個の項目しか含まないからである．したがって，以下では 2 次記憶のことは忘れ，再び **1 レベルの記憶装置**に関する探索木の問題を考えよう．

2分B木 (**BB 木**) は，1 または 2 個の項目をもつ節点 (ページ) から成る．したがって，1 つのページは，子孫へのポインタを 2 または 3 個含む：これが **2-3 木**と呼ばれるゆえんである．B 木の定義に従えば，すべての葉ページは同一のレベルに現われ，また BB 木の葉でないすべてのページは (根の場合も) 2 または 3 個の子孫をもつ．いまは主記憶だけを考えているので，記憶領域を最も経済的に使うことが必須であり，ゆえに，節点中の項目の表現に配列を使うことは不適当と思われる．その代案としては，動的リンク割付けがある；すなわち，各節点内に項目からなる長さ 1 または 2 の連結リストを作る．各節点は高々 3 つの子孫しかもたず，したがって，3 個までのポインタを格納すればよ

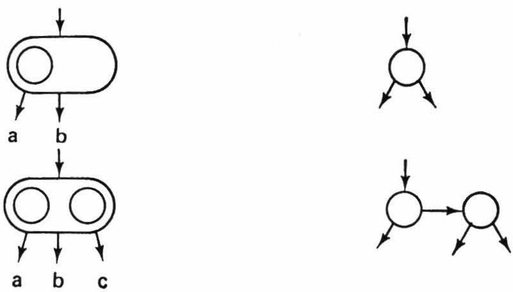


図4.50　BB木の節点の表現

いので，子孫へのポインタと項目リスト中のポインタを**図4.50**のように併用することが考えられる．これにより，B木の節点はその実体を失い，項目は通常の2分木における節点の役目をはたす．しかしながら，子孫(縦方向)へのポインタと，同じページ上の“兄弟”(横方向)へのポインタを区別することが必要なことに変わりはない．右側のポインタだけが横方向であることが許されるので，この区別を記録するには1ビットあれば十分である．したがって，“横方向”の意味をもつ論理型の成分 h を導入する．この表現にもとづいた木節点の定義が(4.84)に与えられている．この方法は，R. Bayer [4-3] によって1971年に提案され研究されたものであり，最大パス長 $p = 2\cdot\lceil \log N \rceil$ を保証する探索木の組織法である．

```
type node = record key: integer;
                   ...........
                   left,right: ref;          (4.84)
                   h: boolean
            end
```

キーの挿入を考えるときには，左または右部分木の成長から生じうる4つの状況を区別しなければならない．この4つの場合が図4.51に描かれている．B木は葉から根に向かって成長する特徴をもっており，またすべての葉は同一のレベルに保持されなければならないことを思い出してほしい．

最も簡単な場合1は，節点Aの**右**部分木が成長し，かつその(仮想的)ページ上にあるキーがAだけの場合である．このときには，子孫Bが単にAの兄弟になる，すなわち縦方向のポインタが横方向のポインタになるだけでよい．

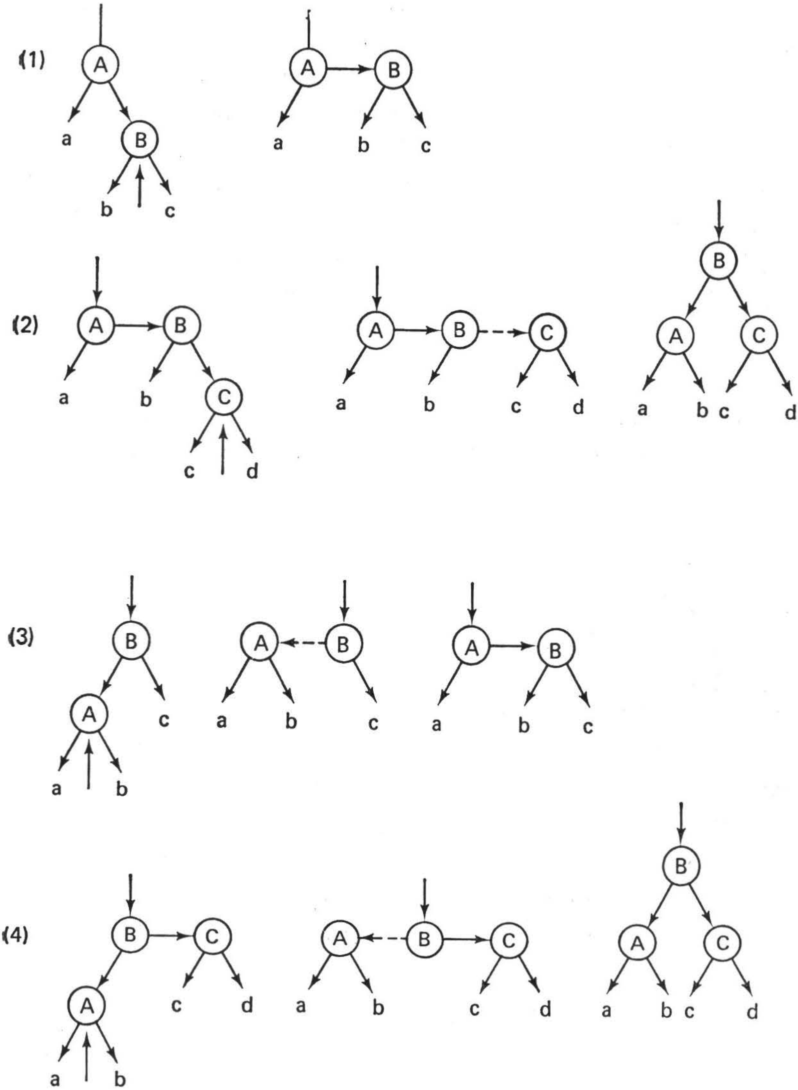


図 4.51　BB 木への節点の挿入

右ポインタのこの単純な“もち上げ”は，Aに既に兄弟がいる場合には不可能である．そのときには，3個の節点からなるページが生じ，したがってそれを分

割しなければならない（場合2）．そして，その中間の節点Bは1段上のレベルに移さなければならない．

次に，節点Bの左部分木の高さが増加したと仮定しよう．再び，Bがページ上の唯一の節点であるときには(場合3)，すなわち右ポインタが子孫へのものであるときには，その左部分木(A)はBの兄弟になることが許される．(左ポインタは横方向であることが許されないので，ポインタの単純な回転が必要である)．しかしながら，Bに既に兄弟がいる場合には，Aのもち上げにより3個の節点をもつページができてしまうので，分割が必要になる．この分割は非常に単純な方法で実現される：CがBの子孫になり，Bは1段上のレベルに上げられる(場合4)．

キーの探索においては，横方向のポインタに沿って進んでいるのか，あるいは縦方向なのかには，実質的差異がないことに注目すべきである．したがって場合3において，そのページが2個以上の要素を含んでいるわけでもないのに，左ポインタが横方向になるのを心配するのは不自然であるように思われる．事実，挿入アルゴリズムは，左および右部分木の成長を扱う際の奇妙な非対称性を明らかにしており，BB木による記憶組織法はむしろ人工的であると考えられる．この組織法の奇妙さを“証明”することはできないが，健全な直感によれば何か“おかしい”ところがあり，この非対称性は取り除かれるべきである．この結果，**対称2分B木（SBB木）**の概念が生まれたが，これもまたBayer[4-4]により1972年に研究されたものである．平均的には，わずかではあるが能率のよい探索木が得られるが，挿入および削除のためのアルゴリズムは少しだけ複雑になる．その上，ポインタの性質を示すために，今度は各節点に2ビット(論理変数 *lh* および *rh*)が必要である．

挿入の問題に限定して詳細な考察を行なうが，そのために再び，部分木の成長に関して4つの場合を区別する．それらは**図4.52**に示されているが，この図からも対称性の得られたことが明らかであろう．兄弟のない節点Aの部分木が成長するときには，常にその部分木の根がAの兄弟になることに注意しよう．したがって，この場合については，これ以上考える必要はない．

図4.52で考えた4つの場合は，すべてページ・オーバフローが起こり，それにひき続いて，ページ分割が行なわれる場合を示したものである．中央の図における横方向ポインタ(3つの兄弟をリンクしている)の向きにしたがって，各場合に名前がつけられている．左側の列には初期状態が示されている．中央の列は，部分木の成長により一段下の節点がもち上げられたことを図示したものである．右側の列の図は，節点の再配置（ページ分割）の結果を示している．

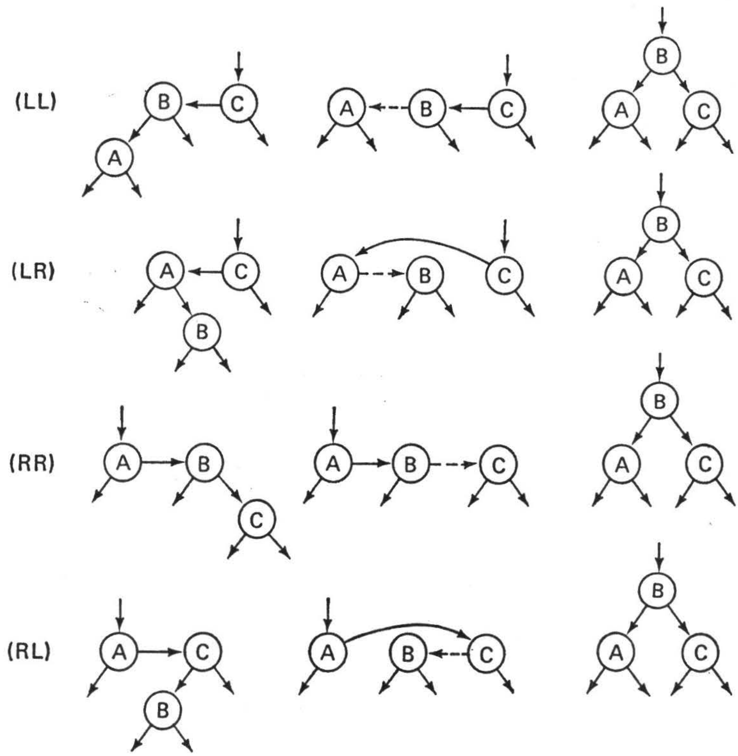


図 4.52 SBB-木への挿入

この組織法を生みだしたページの概念には，これ以上こだわらない方が賢明である．なぜなら，われわれに興味があるのは，最大パス長を$2 \cdot \log N$に制限することだけだからである．このためには，任意の探索パス上で，2つの横方向ポインタが連続することがないことを保証しさえすればよい．しかし，左側および右側への横方向ポインタをもった節点を禁止する理由はない．したがって，次の性質をもつ木として SBB 木を定義しよう：

1. すべての節点は1つのキーと，高々2つの部分木(へのポインタ)をもつ.
2. すべてのポインタは横方向かまたは縦方向である．いかなる探索パス上にも，2つの連続する横方向ポインタはない．
3. すべての端点(子孫を持たない節点)は同一の(終端) レベルに現われる．

この定義より，最長の探索パスは木の高さの2倍より長くはないことがわかる．

N 個の節点からなる SBB 木の高さは，$\lceil \log N \rceil$ 以上になることはないから，$2\lceil \log N \rceil$ は探索パス長の上限であることがただちに導かれる．

このような木がどのように成長するかが**図 4.53** に描かれている．各行は，次のようなキー列の挿入過程のスナップショットを表わしているが，セミコロンはスナップショットのとられた位置に印をつけたものである．

(1)　1　2;　3;　4　5　6;　7;
(2)　5　4;　3;　1　2　7　6;
(3)　6　2;　4;　1　7　3　5;　　(4.85)
(4)　4　2　6;　1　7;　3　5;

これらの図は，B 木の3番目の性質，すなわちすべての端点が同じレベルに現われることを特に明白に表わしている．これらの構造は，はさみで刈り込まれたばかりの庭の垣根に似ているので，**垣根** (hedge) と呼ぶことにする．

垣根木の構成アルゴリズムが (4.87) に定式化されている．それは，節点の型定義 (4.86) に基づいているが，これには左および右ポインタが横方向を指していることを表わす2つの成分 *lh* と *rh* が含まれている．

```
type node = record key: integer;
                   count: integer;
                   left,right: ref;
                   lh,rh: boolean
            end
```
(4.86)

この再帰的手続き *search* も基本的な2分木への挿入アルゴリズムのパターンに従っている (4.87 参照)．3番目のパラメータ h が追加されているが，これは p を根とする部分木が変化したか否かを示しており，B 木探索プログラムにおけるパラメータ h に直接対応するものである．しかしながら，“ページ”を連結リストで表現したことによる結果に注意しなければならない：すなわち，1つのページは1回または2回の探索手続きの呼出しにより走査される．われわれは，（縦方向ポインタによって表わされる）部分木が成長した場合と，（横方向ポインタによって表わされる）兄弟節点が新しい兄弟を得ることによりページ分割が必要になる場合とを区別しなければならない．この問題は，次の意味をもつ3値の h を導入することにより容易に解決される．

1. $h=0$: 部分木 p は木構造の変化を要求しない．

2. $h=1$: 節点 p は兄弟を得た.
3. $h=2$: 部分木 p は高さを増した.

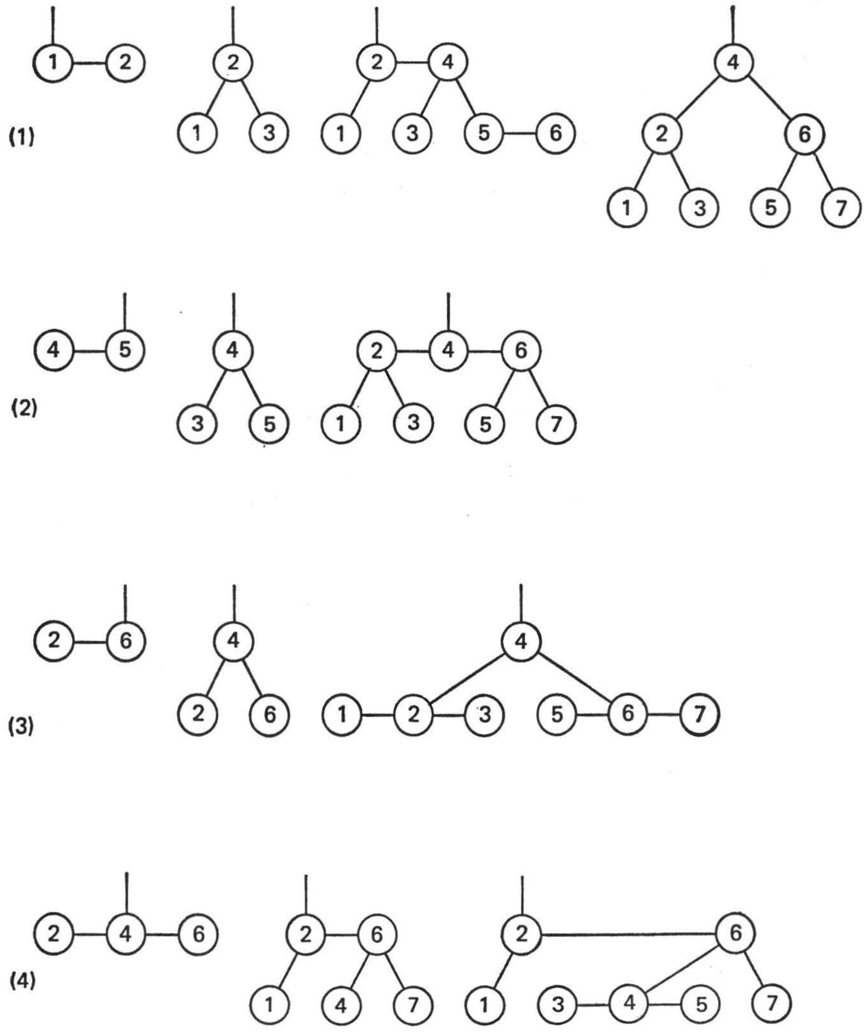


図 4.53 挿入例(4.85)による"垣根"木の成長

```
procedure search(x: integer; var p: ref; var h: integer);
   var p1,p2: ref;
begin
   if p = nil then
   begin {word is not in tree; insert it}
      new(p); h := 2;
      with p↑ do
      begin key := x; count := 1; left := nil;
         right := nil; lh := false; rh := false
      end
   end else
   if x < p↑.key then
   begin search(x,p↑.left,h);
      if h ≠ 0 then
      if p↑.lh then
      begin p1 := p↑.left; h := 2; p↑.lh := false;
         if p1↑.lh then
         begin {LL} p↑.left := p1↑.right;
            p1↑.right := p; p1↑.lh := false; p := p1
         end else                                              (4.87)
         if p1↑.rh then
         begin {LR} p2 := p1↑.right; p1↑.rh := false;
            p1↑.right := p2↑.left; p2↑.left := p1;
            p↑.left := p2↑.right; p2↑.right := p; p := p2
         end
      end else
      begin h := h-1; if h ≠ 0 then p↑.lh := true
      end
   end else
   if x > p↑.key then
   begin search(x,p↑.right,h);
      if h ≠ 0 then
      if p↑.rh then
      begin p1 := p↑.right; h := 2; p↑.rh := false;
         if p1↑.rh then
         begin {RR} p↑.right := p1↑.left;
            p1↑.left := p; p1↑.rh := false; p := p1
         end else
```

```
            if p1↑.lh then
            begin {RL} p2 := p1↑.left; p1↑.lh := false;
               p1↑.left := p2↑.right; p2↑.right := p1;
               p↑.right := p2↑.left; p2↑.left := p; p := p2
            end
         end else
         begin h := h-1; if h ≠ 0 then p↑.rh := true
         end
      end else
      begin p↑.count := p↑.count + 1; h := 0
      end
end {search}
```

節点の再配置のために行なわれる動作が，バランス木の探索アルゴリズム(4.63)において用いられたものに非常によく似ていることに注意してほしい．(4.87)より，4つのすべての場合がポインタの簡単な回転によって実現され得ることは明らかである：*LL* および *RR* の場合には単一の回転が，また *LR* および *RL* の場合には2重の回転が行なわれる．実際には，手続き (4.87) は (4.63) より少しだけ簡単であるように思われる．明らかに，垣根木方式はAVL バランス基準に対する1つの代案になっている．

ここでは，数学的解析にかかわることはせずに，いくつかの基本的な相違点についてのみ述べる．**AVL バランス木は垣根木の部分集合である**ことを証明することができる．したがって，後者のクラスの方が広い．このことから，それらのパス長は平均的には AVL の場合より長い．この点に関して，図4.53の“最悪な場合”の木 (4) に注意してほしい．一方，節点の再配置については，AVL の場合よりその実行回数が少ないであろう．したがって，キーの挿入(あるいは削除)に比べて，キーの検索が非常に頻繁に行なわれるような応用では，バランス木の方が好ましいであろう：もしこの比が中位であるならば，垣根木の方が有利であろう．

この境界がどこにあるかを決定することは非常に困難である．それは，検索の頻度と構造変化のそれとの比だけではなく，実現方法の特性にも強く依存するからである．節点レコードが密に詰められた表現をとっており，したがって成分へのアクセスに部分語の選択が必要になるような場合には，特にそうである．論理型の成分 (垣根木の場合の *lh*, *rh*) は，3値の成分 (バランス木の場合の *bal*) より効率よく扱うことができる場合が多い．

4.6　キー変換(ハッシング)

この最後の節では，次のような一般的問題を用いて，ある動的データ割付け技法の説明を行なう．

> キー値によって特徴づけられる項目の集合 S が与えられ，かつその上で順序関係が定義されているとき，与えられたキー k をもつ項目の検索ができるだけ少ない計算量で行なえるようにするには，S をどのように組織すればよいか．

明らかに，計算機の記憶装置内部では，究極的には，各項目は記憶番地 a を指定することによりアクセスされる．したがって，上で述べられた問題は，キーの集合 K から番地の集合 A への適当な写像 H を見つけることと本質的には同じである．

$$H: K \longrightarrow A$$

4.5節では，この写像は，種々のリストや木の探索アルゴリズムの形で実現されていた．そして，このアルゴリズムは基本となる種々のデータ組織法に依存していた．ここでは，もう1つのアプローチについて述べるが，この方法は，基本的には簡単であり，また多くの場合非常に有効である．これにもいくつかの欠点があるが，これについても述べる．

この技法で用いられるデータ構造は配列構造である．したがって，Hはキーから配列のインデックスへの写像であるが，これがこの技法を一般的に**キー変換**とよぶ理由である．配列は基本的かつ静的なデータ構造のひとつであり，いかなる動的割付け手続きにも依存する必要がないことに注意すべきである．したがって，この節を動的データ構造と題する本章に組み入れるのは，いささか場違いではあるが，木構造と競合するような問題領域にこれが用いられることが多いので，ここで説明するのが適当と思われる．

キー変換を用いる際の基本的な問題点は，起りうるキー値の集合が，利用可能な記憶番地(配列インデックス)の集合に比べて非常に大きいことである．典型的な例としては，たとえば1000人の人間の識別キーとして，アルファベット10文字からなる文字列が使われるような場合がある．このときには，可能なキーが 26^{10} 個もあり，これらは 10^3 個の可能なインデックスに写像されなければならない．したがって，関数 H は多対1写像である．キー k が与えられたとき，検索 (探索) 操作の最初のステップは，対応するインデックス $h = H$

(k)を計算することであり，そして，次の——明らかに必要な——ステップは，キー k をもつ項目が配列(表) T において本当に h よって同定されるか否かを検証すること，すなわち $T[H(k)]=k$ が成立するか否かをチェックすることである．ただちに次の2つの問題が生じる．

1. どのような関数 H を使うべきか．
2. H が目的とする項目の場所を与えない場合には，どのように対処したらよいか．

問題2. に対する答えは，何らかの方法によって，別の場所たとえばインデックス h' を求めることであり，もしこれも求める項目の場所でないならば，さらに h''を求める，$\cdots$ というようにすることである．目的のキー以外のものが同定された場所に存在することは**衝突**(collision) と呼ばれる．そして，別のインデックスを生成する作業は**衝突の処理**といわれる．以下では，変換関数の選択と衝突の処理方法について論じる．

4.6.1 変換関数の選択

良い変換関数のための必要条件は，それがインデックスの値域上にキーをできるだけ一様に分布させることである．たとえこの要求が満たされないとしても，その分布が完全にランダムであるという印象を与えるものであれば実際上は十分であり，特定の分布に限定されることはない．この性質により，この方法は**ハッシング** (hashing)，すなわち“細かく切る”あるいは“めちゃめちゃにする”という，いささか非科学的な名前がつけられており，H は**ハッシュ関数**とよばれている．明らかに，この関数は能率よく計算できる，すなわち非常に少数の基本算術演算によって計算されるものでなければならない．

型変換関数 $ord(k)$ が利用可能であるとし，これが，起りうるすべてのキー集合中におけるキー k の順序数を表わすと仮定する．さらに，配列インデックス i が $0,\cdots,N-1$ の範囲内にあるとする．ただし，N は配列の大きさである．このとき，ひとつの明白な選択は，

$$H(k)=ord(k) \ \mathbf{mod}\ N \tag{4.88}$$

である．この関数は，キー値がインデックスの値域上に一様に広がるという性質を持つので，ほとんどのキー変換の基礎になっている．特に，N が2のべき乗の場合には，これは非常に能率よく計算できる．しかしながら，キーが文字列の場合には，これは避けるべきである．この場合には，すべてのキーが同

様に起こりやすいという仮定は全く誤っている．実際このときには，数文字しか違わない文字列が同一のインデックス上に非常に写像されやすく，したがって，非常に偏りのある分布が生じやすいであろう．一般的には，(4.88)において N が素数であることが特に推奨される[4-7]．この結果，完全な除算が必要になり，単なる2進桁のマスクによってこれに代えることはできないが，現在のほとんどの計算機は除算命令を持っているので，これは重大な欠点ではない．

キーがビット列によって表現されているとき，その一部分に“排他的論理和”のような論理演算を適用するハッシュ関数が使われることもよくある．これらの演算は，計算機によっては，除算より高速であるかも知れないが，インデックスの値域上におけるキーの分布が著しく偏ることがあるので，ここではそのような方法についは，これ以上詳しくは述べない．

4.6.2　衝突の処理

与えられたキーに対応する表中の項目が，求めるものではないという結果になった場合には，衝突が存在する，すなわち，2つのキーが同一のインデックス上に写像されている．この場合には，2次検査が必要であるが，これは与えられたキーだけから決定されるインデックスにもとづくものでなければならない．2次インデックスを生成する方法にはいくつかのものがある．明白かつ有効な1つの方法は，1次インデックス $H(k)$ が同一のすべての項目を連結リストにリンクする方法である．これは，**直接チェイニング**と呼ばれる．このリストの要素は，最初のテーブルの中にあってもよいし，またそうでなくてもよい．後者の場合には，それらが割り付けられる記憶領域は，通常**オーバフロー領域**と呼ばれる．この方法は確かに効果的ではあるが，衝突キーのリストを維持しなければならないし，またテーブルの各行はこの衝突リストへのポインタ（またはインデックス）のための場所を用意しなければならないという欠点をもっている．

衝突を解決するためのもう1つの解は，リンクを全く用いず，その代わり，求める項目が見つかるかまたは空の場所に出合うまで，その同じ表の他の場所を単に調べるというものである．もし空の場所に出合ったならば，その指定されたキーはこの表には存在しないと考える．この方法は**オープン・アドレッシング**（open addressing）とよばれる[4-9]．もちろん2次検査のインデックス列は，与えられたキーに対して常に同一でなければならない．このとき，表引きのアルゴリズムの概略は次のようになる．

```
h := H(k); i := 0;
repeat
    if T[h].key = k then item found else
    if T[h].key = free then item is not in table else
    begin {collision}                                        (4.89)
        i := i+1; h := H(k)+G(i)
    end
until found or not in table (or table full)
```

衝突を解決するためのいろいろな関数が，多くの文献に提案されているが，1968年の Morris [4-8] によるサーベイは，この分野の活動に少なからぬ刺激を与えた．最も簡単な方法は，求めるキーがみつかるかまたは空の場所に出合うまで，次の場所——表が環状であると考えて——を試すことである．したがって $G(i) = i$ であり，この場合，検査に使われるインデックス h_i は，

$$\begin{aligned} h_0 &= H(k) \\ h_i &= (h_0+i) \bmod N, \quad i=1,\ldots,N-1 \end{aligned} \tag{4.90}$$

である．

この方法は，**線形検査法** (linear probing) と呼ばれるが，これには，1次キー(挿入時に衝突を起さなかったキー)の周りにキーが集中しやすいという欠点がある．理想的には，残りの場所にキーが一様に分散するように G を選ぶべきことはもちろんである．しかしながら，実際にはこれは非常に手間がかかるので，計算が簡単でありかつ(4.90)の線形関数よりは優れている妥協案が好まれる．その1つに，2次関数を使う方法があるが，この方法では検査のためのインデックス列は次のように計算される．

$$\begin{aligned} h_0 &= H(k) \\ h_i &= (h_0+i^2) \bmod N \quad (i>0) \end{aligned} \tag{4.91}$$

$h_i = i^2$ と $d_i = 2i+1$ のための再帰公式 (4.92) を用いれば，次のインデックスの計算には2乗演算が必要でないことに注意してほしい．

$$\begin{aligned} h_{i+1} &= h_i+d_i \\ d_{i+1} &= d_i+2 \end{aligned} \qquad (i>0) \tag{4.92}$$

ここで，$h_0 = 0$，$d_0 = 1$ である．この方法は，**2次関数検査法** (quadratic probing) と呼ばれる．必要な計算が簡単であるにもかかわらず，この方法によって1次キーの周りの集中を実質的に避けることができる．

非常に軽微ではあるが，これは次のような欠点をもっている．検査を行なう場合，表中の全ての場所が必ずしも探索されるとは限らない，すなわち，まだ空きが残っているにもかかわらず，挿入時にその空きに出合わない可能性があることである．実際には，もし表の大きさ N が**素数**であるならば，2次関数検査法によって，少なくとも表の**半分**は調べられる．i 番目と j 番目の検査が表の同じ場所で一致したとすると，これは次の方程式によって表わすことができる．

$$i^2 \ \mathbf{mod}\ N = j^2 \ \mathbf{mod}\ N$$

または，

$$(i^2-j^2) \equiv 0 (\mathrm{mod}\ N)$$

差の部分を因数分解すると，

$$(i+j)\,(i-j) \equiv 0\ (\mathrm{mod}\ N)$$

$i \neq j$ であるから，$i+j = cN$（ただし c は整数）であるためには，i または j は $N/2$ 以上でなければならない．

実際には，2次検査や衝突の回避を $N/2$ 回も行なうことは極めて稀であり，表が既にほとんどいっぱいである場合にしか起こらないので，この欠点は重要ではない．

分散記憶技法の応用として，相互参照生成プログラム 4.5 を**プログラム 4.8** の形に書き換える．基本的な差は，手続き *search* とポインタ型 *wordref* を語の表 T で置きかえたことである．ハッシュ関数 H は，表の大きさによる剰余であり，衝突の処理には2次関数検査法が使われている．性能を良くするためには，表の大きさが素数であることが必須である．

プログラム 4.8　ハッシュ表を用いた相互参照の生成

```
program crossref(f,output);
{cross reference generator using hash table}
label 13;
const c1 = 10;      {length of words}
      c2 = 8;      {numbers per line}
      c3 = 6;      {digits per number}
      c4 = 9999;      {max line number}
      p = 997;      {prime number}
      free = '          ';
```

```
type index = 0..p;
     itemref = ↑item;
     word = record key: alfa;
                   first, last: itemref;
                   fol: index
            end ;
     item = packed record
                   lno: 0..c4;
                   next: itemref
            end ;
var i, top: index;
     k,k1: integer;
     n: integer;            {current line number}
     id: alfa;
     f: text;
     a: array [1..c1] of char;
     t: array [0..p] of word;          {hash table}
procedure search;
   var h,d,i: index;
       x: itemref; f: boolean;
   {global variables: t, id, top}
begin h := ord(id) mod p;
   f := false; d := 1;
   new(x); x↑.lno := n; x↑.next := nil;
   repeat
      if t[h].key = id then
         begin {found} f := true;
            t[h] .last↑.next := x; t[h].last := x
         end else
      if t[h].key = free then
         begin {new entry} f := true;
            with t[h] do
            begin key := id; first := x; last := x; fol := top
            end ;
            top := h
         end else
```

訳注 型 *alfa* の宣言がぬけている.

```
            begin {collision} h := h+d; d := d+2;
                if h ≥ p then h := h-p;
                if d = p then
                    begin writeln('TABLE OVERFLOW'); goto 13
                    end
            end
    until f
end {search} ;
procedure printtable;
    var i,j,m: index;
    procedure printword(w: word);
        var l: integer; x: itemref;
    begin write('   ', w.key);
        x := w.first; l := 0;
        repeat if l = c2 then
                begin writeln;
                    l := 0; write('   ':c1+1)
                end ;
            l := l+1; write(x↑.lno:c3); x := x↑.next
        until x = nil;
        writeln
    end {printword} ;
begin i := top;
    while i ≠ p do
    begin {scan linked list and find minimal key}
        m := i; j := t[i].fol;
        while j ≠ p do
            begin if t[j].key < t[m].key then m := j;
                j := t[j].fol
            end ;
        printword(t[m]);
        if m ≠ i then
            begin t[m].key := t[i].key;
                t[m].first := t[i].first; t[m].last := t[i].last
            end ;
        i := t[i].fol
    end
end {printtable} ;
```

```
begin n := 0; k1 := c1; top := p; reset(f);
   for i := 0 to p do t[i].key := free;
   while ¬eof(f) do
   begin if n = c4 then n := 0;
      n := n+1; write(n:c3);      {next line}
      write('  ');
      while ¬eoln(f) do
      begin {scan non-empty line}
         if f↑ in ['A' .. 'Z'] then
         begin k := 0;
            repeat if k < c1 then
               begin k := k+1;  a[k] := f↑;
               end ;
               write(f↑); get(f)
            until ¬(f↑ in ['A' .. 'Z', '0' .. '9']);
            if k ≥ k1 then k1 := k else
            repeat a[k1] := '  ';  k1 := k1−1
            until k1 = k;
            pack(a,1,id); search;
         end else
         begin {check for quote or comment}
            if f↑ = '''' then
               repeat write(f↑); get(f)
               until f↑ = '''' else
            if f↑ = '{' then
               repeat write(f↑); get(f)
               until f↑ = '}' ;
            write(f↑); get(f)
         end
      end ;
      writeln; get(f)
   end ;
13: page; printable
end .
```

キー変換による方法は，この場合には非常に有効である——実際，木構造によるものより能率がよい——が欠点もある．テキストを走査し，語を集めた後で，これらの語をアルファベット順に作表しなければならない．木構造を用いた場合には，これは非常に簡単である．なぜなら，順序つき探索木を基礎にし

ていたからである．しかしながら，キー変換を用いる場合にはそうではない．“ハッシング”という語の意味がこの場面で明らかになる．この場合には，表の印刷を行なう前にソートをしなければならないばかりでなく（簡単のためにプログラム4.8では単純選択ソートを用いた），挿入されたキーを特別なリストにリンクして記憶しておくことが有利になる．したがって，検索だけを考えた場合のハッシング法の優れた性能は，順序のついた相互参照インデックスを作るという全作業を完成させる場合には，必要となる付加的操作によってその一部が相殺されることになる．

4.6.3　キー変換の解析

キー変換による挿入と検索は，最悪な場合にはみじめな振舞を呈することは明らかである．なぜなら探索引数によっては，使用中のすべての場所を検査した後で，目的のもの(または空き場所)が存在しないことがわかるということも確かにあり得るからである．したがって，ハッシュ技法を使う場合には，確率法則の正しさを十分に信用することが必要である．われわれが確信したいことは，平均的には検査の回数は少ないということである．以下の確率論的な議論は，実際それが非常に少ないことを明らかにする．

ここで再び，すべての可能なキーは同様に起りやすいとし，またハッシュ関数 H は，表のインデックスの値域上にそれらを一様に分配すると仮定しよう．さらに，キーは大きさ n の表に挿入されるとし，それには既に k 個の項目が含まれていると仮定する．最初に空の場所が見つかる確率は $1-k/n$ である．これはちょうど比較が1回だけ必要な確率 p_1 に等しい．更にもう1回の検査が必要な確率は，最初の比較において衝突が起る確率と，次に空の場所がみつかる確率の積である．一般に，挿入において i 回の検査が行なわれる確率 p_i は，(49.3)のように求められる．

$$
\begin{aligned}
p_1 &= \frac{n-k}{n} \\
p_2 &= \frac{k}{n}\cdot\frac{n-k}{n-1} \\
p_3 &= \frac{k}{n}\cdot\frac{k-1}{n-1}\cdot\frac{n-k}{n-2} \qquad (4.93)\\
&\cdots \\
p_i &= \frac{k}{n}\cdot\frac{k-1}{n-1}\cdot\frac{k-2}{n-2}\cdot\cdots\cdot\frac{k-i+2}{n-i+2}\cdot\frac{n-k}{n-i+1}
\end{aligned}
$$

したがって，$k+1$ 個目のキーを挿入するときに必要な検査の回数の期待値は，

$$E_{k+1}=\sum_{i=1}^{k+1} i\cdot p_i = 1\cdot\frac{n-k}{n}+2\cdot\frac{k}{n}\frac{n-k}{n-1}+\cdots+(k+1)\cdot \left(\frac{k}{n}\frac{k-1}{n-1}\frac{k-2}{n-2}\cdots\frac{1}{n-k+1}\right)=\frac{n+1}{n-k+1} \quad (4.94)$$

である．ある項目を挿入するのに必要な検査の回数は，それを検索するのに必要な検査の回数に等しいので，結果(4.94)を使って表中の任意のひとつのキーをアクセスするのに必要な検査の平均回数 E を計算することができる．再び，表の大きさを n で表わし，また既に表中にあるキーの数を m とする．このとき，

$$E=\frac{1}{m}\sum_{k=1}^{m}E_k=\frac{n+1}{m}\sum_{k=1}^{m}\frac{1}{n-k+2}=\frac{n+1}{m}(H_{n+1}-H_{n-m+1}) \quad (4.95)$$

ここで，

$$H_n=1+\frac{1}{2}+\cdots+\frac{1}{n}$$

は調和関数である．H_n は，$H_n\cong\ln(n)+\gamma$ のように近似できる（ただし γ はオイラ定数）．さらに，$\alpha=m/(n+1)$ を代入すると次式が得られる．

$$E=\frac{1}{\alpha}(\ln(n+1)-\ln(n-m+1))=\frac{1}{\alpha}\ln\frac{n+1}{n+1-m}$$

$$=\frac{-1}{\alpha}\ln(1-\alpha) \quad (4.96)$$

α は，近似的には，総場所数に対する使用中の場所の数の比であり，**負荷係数**と呼ばれる；$\alpha=0$ ならば表は空であり，また $\alpha=n/(n+1)$ ならば，いっぱ

表 4.6 負荷係数の関数としての検査回数の期待値

α	E
0.1	1.05
0.25	1.15
0.5	1.39
0.75	1.85
0.9	2.56
0.95	3.15
0.99	4.66

いである．ランダムに選ばれたキーを検索あるいは挿入するための検査回数の期待値 E が，**表 4.6** に負荷係数 α の関数として与えられている．この結果の数字は実際驚くべきものであり，キー変換法が極めて性能がよいことを述べている．表が 90% 詰まっていても，キーや空の場所の発見に平均 2.56 回の検査しか必要でない！　特に，この数字が，存在するキーの絶対数にはよらず，ただ負荷係数にのみ依存することに注意してほしい．

上の解析は，衝突の処理において，残りの場所の上にキーが一様に分配されるという仮定の上で行なわれた．実際に用いられる方法では，少し性能が悪くなる．線形検査法の場合を詳細に解析すると，検査回数の期待値 E は (4.97) によって与えられる [4-10].

$$E=\frac{1-\alpha/2}{1-\alpha} \tag{4.97}$$

$E(\alpha)$ のいくつかの数値が**表 4.7** に与えられている．最悪な衝突処理法を用いても，このように良好な結果が得られるので，キー変換(ハッシング)は万能薬であると考えがちである．少なくとも，検索と挿入に必要な比較ステップ数で考えるかぎり，この方法はこれまでに議論してきた最も手の込んだ木構造と比較しても，その性能は優れている．したがって，ハッシングの欠点のいくつかを明確に指摘しておくことは重要であろう．(もちろん偏見のない考え方をすれば，それらは明らかなものではあるが)．

動的割付けを用いた技法と比較した場合の最大の欠点は，明らかに，**表の大きさが固定**していて実際の要求に適合させられないことである．したがって，記憶利用率や性能の低下（さらには表のオーバフロー）を避けようとするならば，類別すべきデータ項目数のかなり良い推定値が**事前**に必要である．項目数が正確にわかっていても(これは極端な場合ではあるが)，良好な性能を望むならば，表は少し（たとえば 10%）大きめにとらなければならない．

表 4.7　線形検査法に対する検査回数の期待値

α	E
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

分散記憶技法の２番目の大きい欠陥は，キーが挿入や検索をされるだけではなく，削除も行なわれなければならない場合に明らかとなる．なぜなら，ハッシュ表からのキーの削除は，別のオーバフロー領域における直接チェイニングでも使わないかぎり，極めて面倒である．したがって，データ量が予測しにくくまたかなり変動し，さらには減少することもあるような場合には，木構造の方がまだ魅力的であるといえるし，実際，好ましいと考えられる．

演習問題

4.1 再帰的型

$$\textbf{rectype}\ T = T_0$$

の概念を，型 T_0 によって定義される値の集合と１個の値 **none** の和集合として定義しよう．すなわち

$$T = T_0 \cup \{\textbf{none}\}.$$

たとえば，型 *ped* の定義［(4.3)参照］は，このとき次のように簡単化される．

$$\begin{aligned}\textbf{rectype}\ ped = \textbf{record}\ & name: alfa;\\ & father, mother: ped\\ \textbf{end}\end{aligned}$$

図 4.2 に対応する再帰的構造の記憶域パターンはどのようになるか．

たぶん，このような機能の実現は，動的記憶割付け方式に基づくであろう．また，上例の *father* と *mother* という名前のフィールドは，プログラマからは見えない自動的に生成されたポインタを含んでいるであろう．このような機能の実現の際の問題点は何か．

4.2 4.2 節の最後の段落で述べたデータ構造をレコードとポインタを用いて定義せよ．上の問題で提案された再帰的型によって，この親族構造を表現することは可能か．

4.3 要素の型が T_0 の**先入れ先出し** (first-in-first-out) 待ち行列 Q を，連結リストによって実現するとしよう．適当なデータ構造を定め，Q に要素を挿入するあるいは抽出する手続き，およびこの待ち行列が空か否かを判定する関数を定義せよ．これらの手続きには，記憶領域の経済的再利用のた

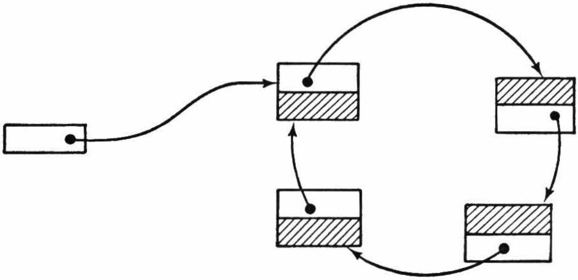


図 4.54　環状リスト

めのメカニズムが組み込まれていなければならない.

4.4　ある連結リスト中のレコードには，型 *integer* のキー・フィールドが含まれているとする．キー値の大きさの順に，このリストをソートするプログラムを書け．次に，このリストを逆転する手続きを作れ.

4.5　環状リスト (**図 4.54** 参照) には，いわゆる**リスト・ヘッダ**がついているのが普通である．そのようなヘッダを導入する理由は何か．与えられたキーによって同定される要素を，挿入，削除，および探索するための手続きを書け．ヘッダが存在する場合およびしない場合の両方について，これを行なえ.

4.6　**2方向性リスト**とは，両方向にリンクされた要素からなるリストである. (**図 4.55** 参照)．ヘッダからも2本のリンクが出ている．前の問題と同様に，要素を探索，挿入，および削除するための1組の手続きを構成せよ.

4.7　入力中に対 $\langle x, y \rangle$ が1つ以上あっても，プログラム 4.2 は正しく動作するか.

4.8　プログラム 4.2 におけるメッセージ "THIS SET IS NOT PARTIAL-

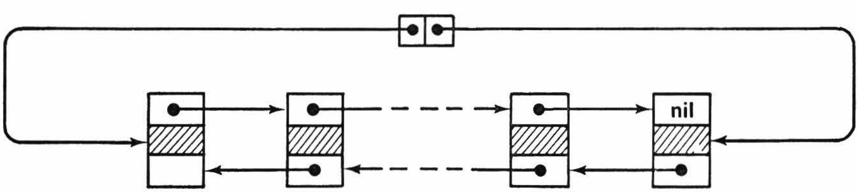


図 4.55　2方向性リスト

LY ORDERED" は，多くの場合それ程役には立たない．このプログラムを拡張して，もしループがあれば，ループを構成している要素の列を出力するようにせよ．

4.9 次のようなプログラムを作れ．プログラム・テキストを読み，すべての手続き(サブルーチン)の定義と呼出しを識別し，サブルーチン間のトポロジカルな順序づけを確立する．ただし，P が Q によって呼ばれるとき $P \prec Q$ と定義する．

4.10 入力が $n+1$ 個の数

$$n, 1, 2, 3, \cdots, n$$

の場合，プログラム 4.3 によって構成される木を描け．

4.11 図 4.23 の木を，先順，中順および後順に走査するときに出合う節点の列はどのようになるか．

4.12 プログラム 4.4 に入力したときに，完全バランス木が得られるような長さ n の数字列の構成法則をみつけよ．

4.13 2分木を走査するための次の2つの順番について考えよ：

(a) (1) 右部分木を走査する．
(2) 根を訪問する．
(3) 左部分木を走査する．

(b) (1) 根を訪問する．
(2) 右部分木を走査する．
(3) 左部分木を走査する．

これらの順番に従って出合う節点の列と，本文中で定義された3つの順番によって生成されるものの間には，簡単な関係があるか．

4.14 n 分木を表現するデータ構造をひとつ定義せよ．次に，この n 分木を走査して同じ要素を含む2分木を生成するプログラムを書け．要素に格納されるキーは k 語を占め，また各ポインタは1語を占めると仮定せよ．n 分木の場合と比べて，2分木を使う場合には記憶領域はどれ位得をするか．

4.15 次のような再帰的データ型にしたがって構成される木を考える(演習問題 4.1 参照)．

rectype *tree*=**record** *x*:*integer*;
left, *right*:*tree*
end

与えられたキー x をもつ要素を見つけ，この要素にある操作 P を施すための手続を定式化せよ．

4.16　あるファイル・システムでは，すべてのファイルのカタログが順序つき2分木として構成されている．各節点はファイルを表わしており，そのファイル名および最後にアクセスした日付けを整数で符号化したものを格納している．

この木を走査して，最後にアクセスされた日付がある特定の日付より前であるすべてのファイルを削除するプログラムを書け．

4.17　ある木構造では，各要素にアクセス回数を付加して，そのアクセス頻度を実際に測定している．ある一定の時間間隔でプログラム 4.4 を使ってこの木を走査し，頻度の降順にキーを挿入することにより，新しい木を生成し木構造の更新を行なっている．この再構成を行なうプログラムを書け．この木の平均パス長は，最適木のそれに比べて，等しいか，劣るか，あるいは非常に劣るか．

4.18　4.5 節で述べた木への挿入アルゴリズムの解析法は，クィック・ソート(プログラム 2.10) によって n 個の要素からなる配列をソートするときの比較および移動(交換)の平均回数 C_n，M_n を計算するためにも使うことができる．ただし，n 個のキー $\{1, 2, \cdots, n\}$ の $n!$ 個のすべての置換は同様に起りやすいと仮定する．類似性を見いだし，C_n と M_n を決定せよ．

4.19　12 個の節点をもったバランス木のうちで，高さが最大のものを描け．アルゴリズム (4.63) によってこの木を生成するためには，どのような順番で節点を挿入しなければならないか．

4.20　手続き (4.63) が再バランス化操作 (LL, LR, RR, RL) を少なくとも1回実行するような n 個の挿入キーの列を見つけよ．そのような列の最小の長さはいくらか．

4.21　手続き (4.63) が4つの再バランス化ルーチンすべてを少くとも1回実行するような n 個のキー $1, \cdots, n$ からなるバランス木およびこれらのキーの置換を見つけよ．長さ n が最小の列は何か．

4.22 フィボナッチ木 T_n の平均パス長はいくらか.

4.23 中心節点を根として選ぶことに基づくアルゴリズム(4.78)に従って，準最適木を生成するプログラムを書け.

4.24 キー 1, 2, 3, …が位数2の空のB木に挿入されるとする（プログラム 4.7）．どのキーによりページ分割が起るか．どのキーにより木の高さが増えるか.

キーが同じ順に削除されるとき，どのキーによりページ統合（と廃棄）が起き，どのキーが高さを減少させるか．(a) バランス化を用いた方式（プログラム 4.7 と同様）と，(b) バランス化を用いない方式（すなわちアンダフローが起きたとき，隣りのページから1つの項目だけをもってくる）の両方の場合について答えよ.

4.25 2分B木にキーを挿入，削除および探索するプログラムを書け．節点の型定義(4.84)を用いよ．挿入方式は，図 4.51 に示されている.

4.26 空の対称2分B木から出発したとき，手続き (4.87) が4つの再バランス化操作(LL, LR, RR, RL)のすべてを少なくとも1回実行するような挿入キーの列をみつけよ.

4.27 対称2分B木から要素を削除するための手続きを書け．次に，4つの再バランス化操作のすべてが少なくとも1回実行されるような木および短い削除列を見つけよ.

4.28 読者の計算機上で，2分木，AVL バランス木，および対称2分B木の挿入と削除の性能を比較せよ．特に，データの詰めこみ，すなわち各節点のバランス情報に2ビットしか用いない経済的なデータ表現の選択の効果について調べよ.

4.29 プログラム 4.6 の印刷アルゴリズムを修正して，横方向および縦方向の枝をもった対称2分B木の表示に利用できるようにせよ.

4.30 各キーの関連情報の量が比較的多い（キー自身と比べて）場合には，この情報はハッシュ表の中に格納すべきではない．この理由を説明し，そのようなデータ集合を表現するための方法を提案せよ.

4.31 オーバフロー・リストの代わりに，オーバフロー木を構成してキーの集中を解決する，すなわち，衝突したキーを木構造に組織する提案について

考察せよ．この場合，分散(ハッシュ)表の各行は，(空のこともある)木の根と考えることができる(木ハッシング)．

4.32 衝突の解決のために2次関数増分を用いるハッシュ表において，挿入と削除を行なう方法を考案せよ．挿入と削除のためのデータとしてランダムなキー列を用いることにより，この方法と単純な2分木構成法との比較を実験的に行なえ．

4.33 ハッシュ表技法の最大の欠点は，表の行数が未知の場合にも，その大きさを固定しなければならないことである．読者の計算機システムには，動的記憶割付け機構が備わっており，いつでも記憶領域を獲得できると仮定する．この場合，ハッシュ表 H がいっぱい（または，ほぼいっぱい）になったときには，もっと大きな表 H' を生成し，H 中のすべてのキーを H' に移し，H のための記憶領域を解放することができる．これは，**再ハッシュ**と呼ばれる．大きさ n の表 H の再ハッシュを行なうプログラムを書け．

4.34 非常に多くの場合，キーは整数ではなく文字列である．これらの文字列は，その長さが一定ではなく，したがって，都合よくまた経済的に固定長のキー・フィールドに格納することはできない．ハッシュ表と可変長キーを操作するプログラムを書け．

参考文献

4-1. Adelson-Velskii, G. M. and Landis, E. M., *Doklady Akademia Nauk SSSR*, **146**, (1962), 263–66; English translation in Soviet Math, 3, 1259–63.

4-2. Bayer, R. and McCreight, E., "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, **1**, No. 3 (1972), 173–89.

4-3. ______, "Binary B-trees for Virtual Memory," *Proc. 1971 ACM SIGFIDET Workshop*, San Diego, Nov. 1971, pp. 219–35.

4-4. ______, "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms," *Acta Informatica*, **1**, No. 4 (1972), 290–306.

4-5. Hu, T. C. and Tucker, A. C., *SIAM J. Applied Math*, **21**, No. 4 (1971) 514–32.

4-6. Knuth, D. E., "Optimum Binary Search Trees," *Acta Informatica*, **1**, No. 1 (1971), 14–25.

4-7. MAURER, W. D., "An Improved Hash Code for Scatter Storage," *Comm. ACM*, **11**, No. 1 (1968), 35–38.

4-8. MORRIS, R., "Scatter Storage Techniques," *Comm. ACM*, **11**, No. 1 (1968), 38–43.

4-9. PETERSON, W. W., "Addressing for Random-access Storage," *IBM J. Res. & Dev.*, **1**, (1957), 130–46.

4-10. SCHAY, G. and SPRUTH, W., "Analysis of a File Addressing Method," *Comm. ACM*, **5**, No. 8 (1962), 459–62.

4-11. WALKER, W. A. and GOTLIEB, C. C., "A Top-down Algorithm for Constructing Nearly Optimal Lexicographic Trees," in *Graph Theory and Computing* (New York: Academic Press, 1972), pp. 303–23.

第5章

言語構造とコンパイラ

本章の目的は，簡単で初等的なプログラム言語のコンパイラ（翻訳プログラム）を作成することである．このコンパイラのプログラムは，自明でない複雑さと大きさのプログラムを，系統的に，構造を整理して構成する実例として役に立つであろう．この点に関して，これは，これまでの各章で明らかにし，また念入りに行なってきたプログラムとデータの構造化法の格好な応用である。さらに，これには，コンパイラの構造と動作への一般的入門という目的もある．この問題に対する知識と洞察力は，高レベル言語によるプログラミング技法の一般的理解を高めるであろうし，また，プログラマが特殊な目的や応用分野に適した彼自身のシステムを作り上げる際に役立つであろう．コンパイラ工学の領域は複雑で広範であることはよく認識されており，したがって，後者の観点における本章の性格は，入門的かつ解説的にならざるを得ない．たぶん最も重要な点は，言語の構造はそのコンパイラの構造に反映されており，その複雑さ（単純さ）はコンパイラの複雑さを強く決定していることである．したがって，まず言語の構成を述べることから始め，次に，単純でまとまりのよい翻訳プログラムが得られるような単純な構造にのみ議論を集中する．構造の単純なこの種の言語構成要素は，偶然にも，実際のプログラム言語において生じるほとんどすべての本質的な要求に対して十分なものである．

5.1 言語の定義と構造

すべての言語は，**語い** (vocabulary) から構成される．その要素は，通常，**語**と呼ばれる．しかしながら，形式言語の分野では，それは（基本）**記号** (symbol) と呼ばれている．語の列のあるものは，その言語の正しい，適切に構成された**文** (sentence) として認められ，また他のものは，正しくない，あるいは適切に構成されていない文と言われる．それでは語の列が正しい文であるか否か

を決定するものは何であろうか．それは**文法**，**構文則**あるいは言語の構造である．実際，文法は（形式的に正しい）文の集合を定義する規則または公式の集合である．しかしながら，さらに重要なことは，そのような規則の集合は，与えられた語の列が文であるか否かの決定を可能にするばかりでなく，文の意味の認識に役立つようなある構造を文に付与していることである．したがって，構文則と意味が密接に関連していることは明らかである．それゆえ，構文的定義は，常に，より高い目的に対して補助的であると考えられる．しかしながら，このことは，初めは構文的側面のみを調べ意味や解釈を無視することを妨げるものではない。

文 "Cats sleep" を考えよう．語 "cats" は主語であり，"sleep" は述語である．この文は，たとえば次の構文則によって定義される言語に属している．

〈文〉::=〈主　語〉〈述　語〉
〈主　語〉::= *cats* | *dogs*
〈述　語〉::= *sleep* | *eat*

これらの3行の意味は，

1. 文は主語の後に述語をつけたものである．
2. 主語は語 "cats" または "dogs" である．
3. 述語は語 "sleep" または "eat" からなる．

この考え方は，**開始記号**〈文〉に**書換え規則**を繰り返し適用することにより文が導かれる，というものである．

これらの規則を書くのに用いたこの定式化または記法は，Backus-Naur-Form (BNF) と呼ばれ，ALGOL 60 の定義に使用されたのがその始まりである．〈文〉，〈主語〉および〈述語〉のような構成要素は**非終端記号**とよばれ，語 cats, dogs, sleep および eat は**終端記号**とよばれ，またこれらの規則は**生成規則**とよばれる．記号 ::= および | はこの BNF 記法のメタ記号である．簡単のために，非終端記号には大文字を，終端記号には小文字を使うことにすると，この例は次のように書き直される．

［例 1］

$$
\begin{aligned}
S &::= AB \\
A &::= x \mid y \\
B &::= z \mid w
\end{aligned} \tag{5.1}
$$

この構文則によって定義される言語は，4つの文 xz, yz, xw, yw からなる．

さらに正確を期するために，次のような数学的定義を述べる：

1. 言語 $L=L(T,N,P,S)$ は次のものにより指定される．
- (a) 終端記号よりなる語い T
- (b) 非終端記号(文法的カテゴリー)集合 N
- (c) 生成規則(構文規則)集合 P
- (d) 開始記号とよばれる(N 中の)記号 S

2. 言語 $L(T,N,P,S)$ は，次の規則3に従って S から生成される終端記号列 ξ の集合である．

$$L=\{\xi \mid S \xrightarrow{*} \xi \quad \text{かつ} \quad \xi \in T^*\} \tag{5.2}$$

(記号列を表わすのにギリシャ文字を用いる)．T^* は T の記号から作られるすべての記号列の集合を表わす．

3. 記号列 σ_n が記号列 σ_0 から**生成**されるとは，次を満たす記号列 σ_1, σ_2, $\cdots, \sigma_{n-1}$ が存在することである．すなわち，すべての σ_i は，下の規則4により σ_{i-1} から直接生成することができる：

$$(\sigma_0 \xrightarrow{*} \sigma_n) \Leftrightarrow (i=1, \cdots, n \text{ に対して} (\sigma_{i-1} \longrightarrow \sigma_i)) \tag{5.3}$$

4. 記号列 η が記号列 ξ から**直接生成**されうるとは，次を満たす記号列 α, β, ξ', η' が存在することである．
- (a) $\xi=\alpha\,\xi'\,\beta$
- (b) $\eta=\alpha\,\eta'\,\beta$
- (c) P は生成規則 $\xi'::=\eta'$ を含む．

注　意：生成規則の集合 $\alpha::=\beta_1$, $\alpha::=\beta_2, \cdots$, $\alpha::=\beta_n$ の略記法として，$\alpha::=\beta_1|\beta_2|\cdots|\beta_n$ を用いる．

たとえば，例1の記号列 xz は，次のような直接生成ステップの列によって生成することができる：$S\rightarrow AB\rightarrow xB\rightarrow xz$；したがって，$S \xrightarrow{*} xz$ であり，また $xz\in T^*$ であるので，xz はこの言語に属する文である．すなわち，$xz\in L$．非終端記号 A と B は最終以外のステップのみに含まれ，一方，最終ステップにおいては，終端記号のみからなる記号列を生成しなければならないことに注意せよ．文法規則が生成規則とよばれる理由は，それらが新しい記号列の生成される仕方を定義しているからである．

言語が**文脈自由型**といわれるのは，それが文脈自由型の生成規則集合によって定義されるときである．そして，生成規則集合が文脈自由型であるとは，そのすべての要素が

$$A ::= \xi \quad (A \in N,\ \xi \in (N \cup T)^*)$$

の形をしている，すなわち，左辺がただ1個の非終端記号だけからなり，A の現われる文脈とは無関係に ξ による置換えが可能なときである．生成規則の形が

$$\alpha A \beta ::= \alpha \xi \beta,$$

ならば，それは**文脈依存型**とよばれるが，その理由は，ξ による A の置換えが α と β という文脈においてのみ起こりうるからである．以下では，文脈自由な場合についてのみ考えることにする．

例2は，再帰によって有限個の規則から無限個の文が生成されうる方法を示している．

［例 2］

$$\begin{aligned} S &::= xA \\ A &::= z \mid yA \end{aligned} \tag{5.4}$$

開始記号 S から次のような文が生成可能である．

$$\begin{array}{l} xz \\ xyz \\ xyyz \\ xyyyz \\ \cdots\cdots\cdot \end{array}$$

5.2 文の解析

言語の翻訳あるいは処理システムの主な仕事は，文や文構造の生成ではなく，その**認識**である．これは，文を読むことにより，その文を導いた生成ステップを再構成することである．すなわち，生成ステップの逆をたどることである．これは，一般には非常に複雑な作業であり，場合によっては不可能なことさえある．その複雑さは，言語を定義するのに使われた生成規則の種類に強く依存するが，構文規則の複雑な言語に対する認識アルゴリズムを構成することは，**構文解析**理論の仕事である．ここでは，実際に役立つような簡単で能率のよい

認識アルゴリズムを作る方法について概説する．このようなアルゴリズムにおいては，文の解析に必要な計算は，その長さの線形関数でなければならず，最悪な場合でも，この関数関係は $n \cdot \log n$ でなければならない．ここで，n は文の長さである．与えられた任意の言語に対する認識アルゴリムを見つけるという問題を考える代わりに，ここでは，現実的な逆の立場をとる．すなわち，能率のよいアルゴリズムを定義し，次に，それによって扱いうる言語のクラスを決定する [5-3]．

基本的な能率上の要求から，各解析ステップは，現在の計算状態と次に読み込まれる1つの記号にのみ依存するものでなければならないことが，まず結論される．もうひとつの重要な要求は，いかなるステップも以後取り消されることがない，というものである．これらの2つの要求は，通常**バックトラックのない1記号の先読み**とよばれている．

ここで説明する基本的な方法は，**トップダウン**解析（または下降解析）とよばれるが，その理由は，生成ステップ(一般には構造木になる)の再構成を，開始記号から最終的な文に向かって，すなわち頂上(根)から下(葉)に向かって行なおうと試みるからである[5-5 および 5-6]．例1について再び考えよう：文 *Dogs eat* が与えられ，この文がこの言語に属するか否かを決定しなければならないとする．定義から，このようなことが起こるのは，開始記号〈文〉からそれが生成される場合だけである．構文則より，それが文であるのは，主語の後に述語が続いた形をしている場合だけであることは明らかである．したがって，残りの仕事を2つに分ける：まず最初に，この文の先頭部分が，記号〈主語〉から生成されるかどうかを決定する．これは正しい．なぜなら，*dogs* は直接生成可能だからである：記号 *dogs* は入力文中で検査済みとなり（したがって，読取り位置を進め），われわれは次の仕事に進む：すなわち，残りの部分が記号〈述語〉から生成できるか否かを検査する．これは再び正しいので，解析過程は成功する．この過程は次のように図示することができるが，その左側は未処理の作業を，また右側の部分は，未だ読み込まれていない入力を表わしている．

〈文〉		*dogs eat*
〈主語〉	〈述語〉	*dogs eat*
dogs	〈述語〉	*dogs eat*
	〈述語〉	*eat*
	eat	*eat*
	—	—

次の図は，例2の生成規則に従って文 $xyyz$ を解析する過程を示したものである．

S	$xyyz$
xA	$xyyz$
A	yyz
yA	yyz
A	yz
yA	yz
A	z
z	z
—	—

文の生成ステップを逆にたどる過程は，**構文解析**とよばれるので，上に述べたものは**構文解析アルゴリズム**である．この2つの例では，入力記号列中の次の1記号を調べることにより，個々の置換えステップを決定することができた．残念なことに，次の例からも明らかなように，これは常に可能というわけではない．

［例 3］

$$\begin{aligned} S &::= A \mid B \\ A &::= xA \mid y \\ B &::= xB \mid z \end{aligned} \tag{5.5}$$

文 $xxxz$ の解析を行なってみると，

S	$xxxz$
A	$xxxz$
xA	$xxxz$
A	xxz
xA	xxz
A	xz
xA	xz
A	z

となり，行き詰まってしまう．この問題点は，最初のステップにおいて，S を A または B のどちらで置き換えるかの決定が，最初の1記号を調べただけでは不可能なことによるものである．この解決策としては，可能な場合の1つを

仮に調べてゆき，もうそれ以上進めなくなった時点で，これまで調べてきた道を後戻りすることが考えられる．この動作は，**バックトラック**とよばれる．例3の言語においては，“取消”されなければならないステップ数に限界がなく，明らかに，これは最も望ましくない状況である．したがって，バックトラックを生じるような言語構造は，実際の応用においては避けなければならない．ゆえに，これからは次のような制限を満たす文法だけを考えることにする．この制限は，同一の左辺をもつ生成規則において，その右辺の先頭記号の集合がすべて異なることを述べたものである．

規則 1

生成規則

$$A ::= \xi_1 | \xi_2 | \cdots | \xi_n$$

が与えられたとき，ξ_i から生成される文の先頭の記号の集合はすべて異なっていなければならない．すなわち，すべての $i \neq j$ に対して，

$$first(\xi_i) \cap first(\xi_j) = \phi$$

集合 $first(\xi)$ は，ξ から生成される文の先頭に現われるすべての終端記号の集合である．この集合は，次の規則に従って計算される：

1.　引数の初めの記号が終端記号の場合：

$$first(a\xi) = \{a\}$$

2.　初めの記号が，

$$A ::= \alpha_1 | \alpha_2 | \cdots | \alpha_n$$

のような生成規則をもつ非終端記号の場合：

$$f(A\xi) = first(\alpha_1) \cup first(\alpha_2) \cup \cdots \cup first(\alpha_n)$$

例3においては，$x \in first(A)$ かつ $x \in first(B)$ であり，したがって規則1に反している．例3の言語に対する構文則で，規則1を満たすものを見つけることは簡単である．その解は，すべての x が処理されるまで分岐を遅らせることにある．次の生成規則は，それが (5.5) と同一の文集合を生成するという意味で，(5.5) の生成規則と**等価**である：

$$\begin{aligned} S &::= C | xS \\ C &::= y | z \end{aligned} \tag{5.5a}$$

残念なことに，規則1は，別のトラブルが避けられる程強力ではない．次の例を考えよう．

［例 4］

$$S ::= Ax$$
$$A ::= x \mid \epsilon \tag{5.6}$$

ここで，ϵ は空記号列を表わす．文 x を解析しようとすると，次のような“袋小路”に入る可能性がある：

S	x
Ax	x
xx	x
x	$-$

このトラブルの原因は，$A ::= \epsilon$ の代わりに生成規則 $A ::= x$ を用いたことにある．この状況は，**空記号列問題**とよばれるが，これは，空記号列を生成できるような非終端記号についてのみ起りうる．これを避けるために，次の規則を仮定する．

規則 2

空記号列を生成できる記号 $A \in N$ に対しては $(A \xrightarrow{*} \epsilon)$，その先頭記号集合は，$A$ から生成される記号列の後に続く記号の集合と共通部分があってはならない．すなわち，

$$first(A) \cap follow(A) = \phi$$

集合 $follow(A)$ を計算するには，

$$X ::= \xi A \eta$$

の形のすべての生成規則 P_i について考え，$S_i = first(\eta_i)$ を計算し，そのような集合 S_i すべての和集合を求めればよい．もし，空記号列を生成できる η_i が1つでもあれば，$follow\ (X_i)$ も $follow\ (A)$ に含めなければならない．例4においては，記号 A に関してこの規則2が守られていない．なぜなら，

$$first(A) = follow(A) = \{x\}$$

記号の繰返しのパターンを表現する普通の方法は，文の構成に関する再帰的定義を用いることである．たとえば，生成規則

$$A ::= B \mid AB$$

は，記号列集合 B, BB, BBB, $\cdots$ を表現する．しかしながら，このような生成規則は，規則 1により禁止される．なぜなら，次が成立するからである．

$$first(B) \cap first(AB) = first(\mathrm{B}) \neq \phi$$

もし，この生成規則を，少し修正したもの

$$A ::= \epsilon \mid AB$$

で置き換え，ϵ, B, BB, BBB, $\cdots$ を生成するようにすると，今度は規則 2に反する．なぜなら，

$$first(A) = first(B)$$

であり，したがって，

$$first(A) \cap follow(A) \neq \phi$$

となるからである．明らかに，この2つの制約規則は左再帰的定義を禁止している．これらの形を避けるための簡単な方法は，右再帰的定義

$$A ::= \epsilon \mid BA$$

を用いるか，BNF 記法を拡張し，繰返しを陽に表現することを許すことである：われわれはそのような拡張を許し，記号列集合

$$\epsilon,\ B,\ BB,\ BBB, \cdots$$

を {B} で表わすことにする．もちろん，このような構文は，常に空記号列を生成できることに注意しなければならない．(括弧 { と } は拡張されたBNF記法のメタ記号である.)

以上の議論および生成規則(5.5)から(5.5a)への変換により，文法の変換という"手品"は，構文解析に関するどの問題に対しても有効な万能薬であるように思えるかもしれ知れない．しかしながら，文の構造は文の意味を定義するための道具であり，構成された文の意味は，通常その構成要素の意味によって表現されることを考えておかなければならない．たとえば，オペランド a, b, c と引算を表わすマイナス記号 − からなる式の集合を考えよう．

$$S ::= A \mid S - A$$
$$A ::= a \mid b \mid c$$

この文法に従えば，文 $a-b-c$ は，$((a-b)-c)$ のように括弧を用いて表わさ

れる構造をもっている．一方，もしこの文法を変換して，構文的には等価であるが左再帰のない形

$$S ::= A \mid A{-}S$$
$$A ::= a \mid b \mid c$$

にしたとすると，この同じ文には異なった構造，すなわち，$(a-(b-c))$ のように表わされるものが与えられる．引算の通常の意味から考えると，この2つの形は意味的には決して等価ではないことがわかる．

したがって，固有の意味をもった言語を定義する場合には，構文則の定義においても意味構造に常に気をつけていなければならない．なぜなら，後者は前者を反映しているはずだからである．

5.3 構文グラフの構成

前節では，制約規則1と2を満たす文法に対して適用可能なトップダウン認識アルゴリズムについて述べた．ここでは，このアルゴリズムを具体的なプログラムに直す問題を考える．このとき，適用可能な方法としては，本質的に異なる2つのものがある．ひとつは，（規則 1と2を満たす）すべての文法に対して有効な一般的なトップダウン解析プログラムを設計することである．この場合，個々の文法は適当なデータ構造の形で与えられ，これに基づいてこのプログラムは動作する．この一般的な構文解析プログラム（以後，**パーザ**(parser)とよぶ）は，このデータ構造によって制御されているとも考えられるので，**表制御型**と呼ばれる．もう1つの方法は，与えられた言語専用のトップダウン型パーザを作り上げるものである．これは，与えられた構文則をステートメントの列すなわちプログラムに写像する規則の集合に従って，パーザを系統的に構成するものである．いずれの手法にも，それぞれ長所，短所があるが，以後順次これらについて述べる．与えられたプログラム言語に対するコンパイラを作成する場合には，一般的なパーザの高度な柔軟性とパラメータ化はほとんど不必要である．これに反し，個別パーザによるアプローチによれば，より能率のよい，扱いやすいシステムが得られるのが普通であるので，この手法の方が望ましい．いずれの場合においても，与えられた構文を，いわゆる**認識グラフ**または**構文グラフ**によって表現するのが便利である．このグラフは，文の解析過程における制御の流れを反映している．

トップダウン法の特徴は，解析過程の**目標**が最初の時点において既知なことである．この目標は，文，すなわち，開始記号から生成される記号列を認識す

ることである．生成規則の適用，すなわち記号列による記号の置換えは，1つの目標をいくつかの部分目標に分割することであり，これらの部分目標は一定の順序で調べられる．したがって，トップダウン法は，**目標志向型**解析とよばれる．パーザを構成する場合に，非終端記号と目標の間のこの明らかな対応関係を利用することは簡単である：すなわち，各非終端記号に対して，ひとつの部分パーザを作ればよい．各部分パーザの目標は，それに対応する非終端記号から生成される部分文を認識することである．われわれの目的は，パーザ全体を表現するグラフを構成することであるので，各非終端記号は，ひとつの部分グラフに写像されるであろう．以上より，認識グラフを構成するための次のような規則が得られる．

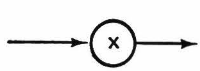
グラフ構成規則

A1.　生成規則の集合

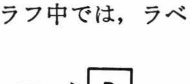
$$A ::= \xi_1 | \xi_2 | \cdots | \xi_n$$

をもつどのような非終端記号 A も，ある認識グラフ A に写像されるが，その構造は，この生成規則の右辺により，規則 $A2$ から $A6$ に従って決定される．

A2.　ξ_i 中の終端記号 x の出現は，この記号に対する認識ステートメントに対応し，読取りプログラムを入力文中の次の記号に進めることを意味する．これはグラフ中では，ラベル x を円で囲んだ枝によって表現される．



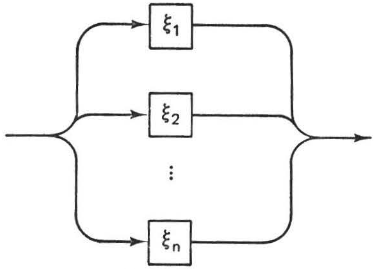
A3.　ξ_i 中の非終端記号 B の出現は，認識プログラム B を起動することに対応する．これはグラフ中では，ラベル B の枝によって表現される．



A4.　次の形の生成規則

$$A ::= \xi_1 | \cdots | \xi_n$$

は，グラフ



に写像される．

ここですべての $\boxed{\xi_i}$ は，構成規則 A2 から A6 を ξ_i に適用することにより得られる．

A5.　次の形の ξ，

$$\xi=\alpha_1\alpha_2\cdots\alpha_m$$

は，グラフ



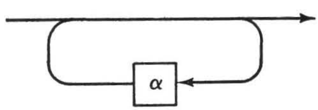
に写像される．

ここで，すべての $\boxed{\alpha_i}$ は，構成規則 A2 から A6 を α_i に適用することにより得られる．

A6.　次の形の ξ

$$\xi=\{\alpha\}$$

は，グラフ



に写像される．

ここで，$\boxed{\alpha}$ は，規則 A2 から A6 を α に適用することにより得ら

れる.

[例　5]

$$
\begin{aligned}
A &::= x \mid (B) \\
B &::= AC \\
C &::= \{+A\}
\end{aligned}
\tag{5.7}
$$

ここで+，x，(，および)は終端記号であり，また{と}は拡張BNFのメタ記号である．Aから生成可能な言語は，オペランドx，オペレータ+および括弧から作られる式の集合である．文の例としては，次のようなものがある．

$$
\begin{aligned}
&x \\
&(x) \\
&(x+x) \\
&((x)) \\
&\cdots\cdots
\end{aligned}
$$

6つの構成規則の適用により得られたグラフが**図5.1**に示されている．CをBの中に，またBをAの中に代入することにより，これらの1組のグラフは1個のグラフに縮約可能であることに注意せよ(**図5.2**参照)．

認識グラフは，その言語の文法と等価な表現であり，BNFの生成規則の集合の代わりに使うことができる．それは非常に便利な形式であり，(ほとんどで

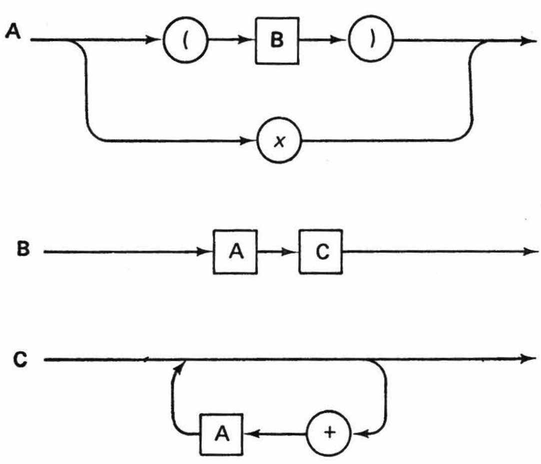


図5.1　例5に対応する構文グラフ

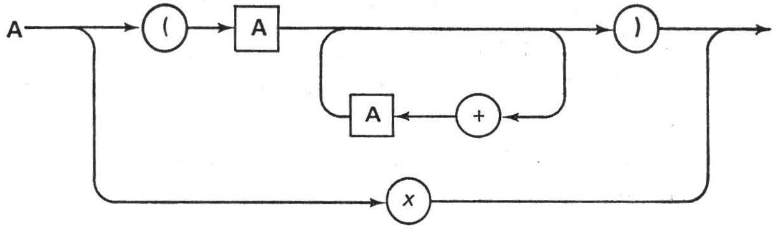


図 5.2 例 5 に対応する縮約構文グラフ

はないにしても）多くの場合 BNF より好ましい．なぜなら，言語の構造をより明確に，より簡潔に描くことができるし，また解析過程をより直接的に理解できるからである．グラフは，言語の設計者のための適切な形式である．言語全体の構文記述の例としては，5.7 節に PL/0 が，また補遺 II に PASCAL が与えられている．

制約規則 1 と 2 は，1 記号の先読みによって，解析が決定性になるように課せられたものであった．これらの規則は，グラフ表現ではどのようになるであろうか．グラフ表現の明快さが最も明らかになるのは，この点に関してである：

1. 規則 1 は，分岐点における枝の選択が，その枝上の次の記号を調べることによってのみ行なわれる，という要求に翻訳される．これは，同一の記号で始まるような 2 本の枝は存在しないことを意味する．
2. 規則 2 は，もし，グラフ A が入力記号を全く読まずにたどれるならば，この "空な枝" は，A の後に続く可能性のあるすべての記号によってラベルづけられなければならない，という要求に翻訳される．（これは，この枝に入る際に行なわれる決定に影響を及ぼす）．

1 組のグラフが，これら 2 つの翻訳された規則を満たすか否かを検証することは簡単であり，その文法の BNF 表現に戻る必要はない．補助ステップとして，集合 $first(A)$ と $follow(A)$ を各グラフ A に対して決定すれば，規則 1 と 2 がただちに適用できる．この 2 つの規則を満足する 1 組のグラフを，われわれは**決定性構文グラフ**とよぶ．

5.4 与えられた構文則に対するパーザの構成

ある言語を受理あるいは解析するプログラムは，その決定性構文グラフ（も

しそのようなグラフが存在すれば）からただちに導かれる．このグラフは，本質的には，そのプログラムのフロー・チャートを表わしている。しかしながら，このプログラムを構成するときには，一定の翻訳規則に厳密に従って行なうのがよい．これらの規則は，初めに BNF から構文則のグラフ表現を得たときのものに似ている．これらの規則を以下に挙げるが，それらはある特定な環境のもとでのみ適用可能である．この環境は，種々の部分目標に対応する手続きが埋め込まれた主プログラムと，次の記号に進むためのルーチンから構成されている．

簡単のために，解析すべき文はファイル *input* によって表現され，また終端記号は個々の文字であると仮定しよう．さらに，この環境の中には，次に読むべき記号を常に表わしている文字型変数 *ch* が存在すると仮定する．この場合，次の記号に進むことは，次のステートメントにより行なわれる．

$$read\,(ch)$$

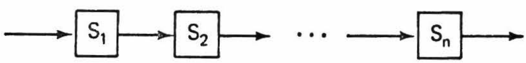
主プログラムは，最初の文字を読むための初期ステートメントの後に，解析の主目標を起動するステートメントが続いたものである．解析目標またはグラフに対応する個々のルーチンは，次の規則に従って得られる。以下では，グラフ S を翻訳して得られるステートメントを $T(S)$ で表わす．

グラフからプログラムへの翻訳規則

B 1.　適当な代入により，グラフの個数ができるだけ少なくなるように縮約を行なう．

B 2.　以後の規則 B3 から B7 に従って，各々のグラフを手続き宣言に翻訳する．

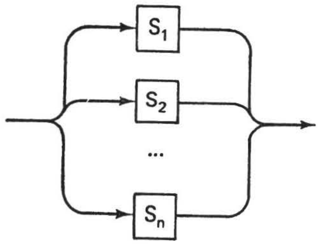
B 3.　要素の列



は，次の複合文に翻訳される．

$$\textbf{begin}\ T(S_1)\ ;\ T(S_2)\ ;\ \cdots\ ;\ T(S_n)\ \textbf{end}$$

B 4.　要素の選択



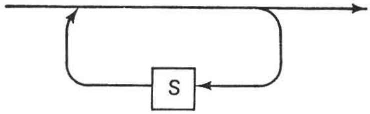
は，次の選択文または条件文に翻訳される．

case *ch* **of** L_1 : $T(S_1)$; L_2 : $T(S_2)$; …… L_n : $T(S_n)$ **end**	**if** *ch* **in** L_1 **then** $T(S_1)$ **else** **if** *ch* **in** L_2 **then** $T(S_2)$ **else** ……… **if** *ch* **in** L_n **then** $T(S_n)$ **else** *error*

ここで，L_i は構文要素 S_i の最初の記号の集合を表わす（$L_i = first(S_i)$）．

注　意：もし L_i が 1 個の記号 a だけから成るときには，もちろん "*ch* **in** L_i" は "$ch = a$" と表わすべきである．

B 5.　次の形のループ

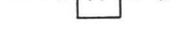


は，ステートメント

while *ch* **in** L **do** $T(S)$

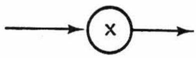
に翻訳される。ここで，$T(S)$ は規則 B3 から B7 に従って S を翻訳したものであり，集合 L は $L = first(S)$ である（直前の注意参照）．

B 6.　他のグラフ A を表わすグラフ要素



は，手続き呼出しステートメント A に翻訳される．

B 7.　終端記号 x を表わすグラフ要素



は，次のステートメントに翻訳される．

if *ch* = *x* **then** *read* (*ch*) **else** *error*

ここで *error* は，誤った構造に出合ったときに呼び出されるルーチンである．

例5の縮約されたグラフ（図 5.2）に，これらの規則を適用して得られる認識プログラムを次に示す（**プログラム 5.1**）．この翻訳の過程において，プログラムを簡単にするために，いくつかのプロラミング上の明らかな規則を自由に用いた．たとえば，機械的に翻訳を行なうと，4行目は，

プログラム 5.1　例5の文法に対する構文解析プログラム

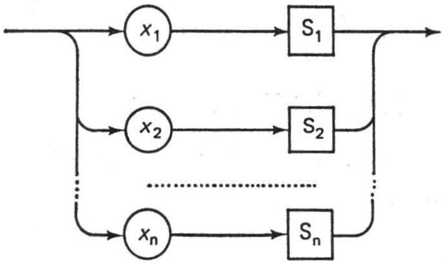
```
program parse (input, output);
    var ch: char;
    procedure A;
    begin if ch = 'x' then read(ch) else
            if ch = '(' then
            begin read(ch); A;
                while ch = '+' do
                    begin read(ch); A
                    end ;
                if ch = ')' then read(ch) else error
            end else error
    end ;
begin read(ch); A
end
```

```
if ch = 'x' then
      if ch = 'x' then read(ch) else error
else ....
```

のようになるであろうが，明らかに，これはプログラムに表わされているような，より簡単な形に変形することができる．また，5行目と7行目の入力ステートメントも同様な簡単化により得られたものである．

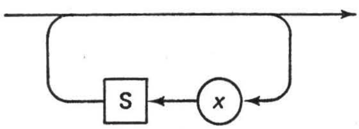
一般に，そのような簡単化がどのような場合に可能であるかを調べ，それをグラフの用語で直接表現することは有意義であるように思われる．次の付加的な2つの規則はそのような例である．

B4a




```
if ch = 'x1' then begin read(ch) ; T(S1) end else
if ch = 'x2' then begin read(ch) ; T(S2) end else
      ............
if ch = 'xn' then begin read(ch) ; T(Sn) end else error
```

B5a




```
while ch = 'x' do
   begin read(ch); T(S) end
```

さらに，頻繁に起る次のような構造

$$read(ch)\,;T(S)\,;$$
while B **do**
　　begin $read(ch)$; $T(S)$ **end**

は，もちろん，より短い形式

repeat $read(ch)$; $T(S)$ **until not** B (5.8)

によって表現することができる．

手続き *error* については，これまで意識的に何も述べないできた．現在われわれに興味があるのは，入力列が正しく構成されているか，あるいは，誤って構成されているかだけであるので，この手続きはプログラムを停止させるものと考えておけばよい．もちろん実際には，もっと手の込んだ方法を使って，誤って構成された文の処理を行なわなければならないが，これについては，5.9節で述べる．

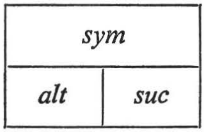
5.5　表制御型構文解析プログラムの構成

個々の言語や構文則に対する個別のプログラムを，前節で与えた規則に従って構成する代わりに，1つの汎用構文解析プログラムを作ることも可能である．この場合，個々の言語の文法は，解析すべき文に先立って，初期データの形でこの汎用プログラムに与えられる．この汎用プログラムは，単純なトップダウン解析法の規則に厳密に従っており，したがって，もし基礎となる構文グラフが決定性，すなわち，1記号の先読みによりバックトラックなしに構文解析が可能なような文法であるならば，その構造は簡単である．

ゆえに，文法（決定性構文グラフの集合の形で表現されていると仮定する）はプログラムの構造にではなく，適当なデータ構造に翻訳される [5-2]．グラフを表現するための自然な方法は，各記号に対して1つの節点を導入し，ポインタによってこれらを結合することである．したがって．この“表”は単なる配列ではない．翻訳のための規則が以下に与えられているが，説明の必要はないであろう．データ構造の節点は，2つの変形をもつ可変レコードである：そのひとつは終端記号のためであり，他は非終端記号のためである．前者は，それの表わす終端記号により同定され，また後者は，その非終端記号を表わすデータ構造へのポインタにより同定される．どの変形も2つのポインタを含んでいるが，ひとつは次に続く記号を示す *successor* であり，もうひとつは，可能な選択を示す *alternative* である†．以上より，(5.9) で定義されるデータ構

† 訳注　(5.9) では，それぞれ *suc*, *alt* と略記されている．

造が得られるが，グラフ中では節点は次のように描かれる．



空列を表わすための要素，空記号も必要になるので，*empty* と呼ばれる終端記号によってそれを表わすことにする．

```
type pointer = ↑node;
  node =
  record suc,alt: pointer;
     case terminal: boolean of
        true: (tsym: char);
        false: (nsym: hpointer)
  end
```
(5.9)

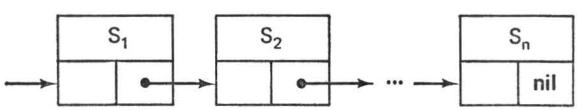
グラフからデータ構造への翻訳規則は，規則 **B1.** から **B7.** に類似している．

グラフからデータ構造への翻訳規則

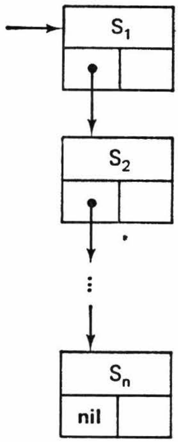
C 1.　適当な代入により，個々のグラフの数ができるだけ少なくなるように縮約を行なう．

C 2.　以後の規則 **C3.** から **C7.** に従って，各々のグラフをデータ構造に翻訳する．

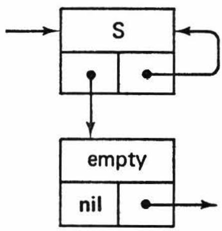
C 3.　要素の列（規則 **B3.** の図参照）は，データ節点からなる次のリストに翻訳される．



C 4.　選択構造（規則 **B4.** の図参照）は，次のデータ構造に翻訳される．



C 5.　ループ（規則 **B 5.** の図参照）は，次の構造に翻訳される．



たとえば，例5の構文則に対応するグラフ（図5.2)からは，**図 5.3** の構造が得られる．このデータ構造は，**ヘッダー** (*header*) 節点によって同定されるが，これにはこの構造が表わす非終端記号（目標）の名前が含まれる．このヘッダーは，今までは必要でなかったが，その理由は目標 (*goal*) フィールドのポインタが，指定された構造の“入口”を直接指すことも可能であったからである．ヘッダーは，しかし，構造の印刷用の名前を格納するために使うことができる．

$$
\begin{array}{l}
\textbf{type}\ hpointer = \uparrow header;\\
\quad header =\\
\qquad \textbf{record}\ entry\colon pointer;\\
\qquad\qquad sym\colon char\\
\qquad \textbf{end}
\end{array}
\qquad (5.10)
$$

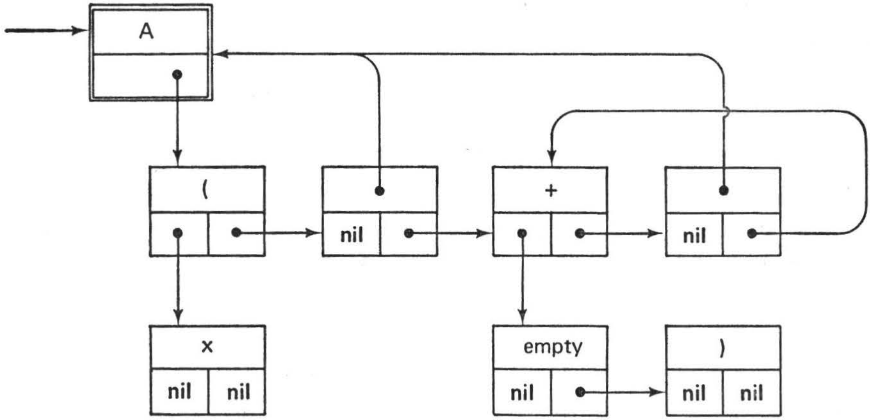


図 5.3　図 5.2 のグラフを表現しているデータ構造

文——入力ファイル上の文字列として表現されている——を解析するためのプログラムは，節点から次の節点への遷移を表わすステートメントの繰返しによって構成される．このプログラムは，グラフの解釈を記述する手続きとして表現されており，非終端記号を表わす節点に出合うと，現在のグラフの解釈の完了に先立って，そのグラフの解釈を実行する．したがって，この解釈手続きは，再帰的に起動される．もし，入力ファイル中の現在の記号 (*sym*) が，デ

```
procedure parse(goal: hpointer: var match: boolean);
    var s : pointer;
begin s := goal↑.entry;
    repeat
        if  s↑.terminal then
            begin if s↑.tsym = sym then
                    begin match := true; getsym
                    end
                else match := (s↑.tsym = empty)
            end
        else parse(s↑.nsym, match);
        if match then s := s↑.suc else s := s↑.alt
    until s = nil
end                                                        (5.11)
```

ータ構造中の現在の節点に含まれる記号と一致すれば，次のステップを示すものとして *suc* フィールドが選ばれ，そうでなければ，*alt* フィールドが選ばれる．

この構文解析プログラムは，新しい部分目標 G が現われると，ただちにそれを追跡し，現在の記号が先頭記号集合 $first(G)$ に含まれるか否かの検査は行なわない．このことは，適合する非終端記号に関して，対象となる構文グラフに一意性がなければならないことを意味している．特に，空記号列を生成できる非終端記号がある場合には，その右辺のどれも非終端記号で始まってはいけない．

もっと制限の緩い文法のクラスを対象とするような，より複雑な表制御型パーザを(5.11)から導くことが可能である．多少の変更により，バックトラックをすることもできるが，能率は著しく低下する．

グラフによる文法の表現には，ひとつの決定的な欠点がある：すなわち，計算機が直接にはグラフを読むことができないことである。一方，構文解析を始める前には，何らかの方法によりパーザを制御する表を構成しなければならない．この点に関しては，BNF による文法の表現は，汎用パージング・プログラムの入力形式として理想的であるように思われる．したがって，次節では BNF 生成規則の列を入力し，それらを規則 B1 から B6 に従って内部データ構造に変換し，パーザ(5.11)が動作できるようにするためのプログラムについて述べる[5-8]．

5.6 BNF からパーザ制御用データ構造への変換

BNF 生成規則を入力し，それらを他の表現に変換する翻訳プログラムは，ある言語に属すると考えられる文を入力データとするようなプログラムの格好な例である．実際，BNF 自身それ固有の構文則によって特徴づけられる言語と考えるのが最も適当である．もちろんこの構文則も，再び，BNF 生成規則によって定義することができる．その結果，この翻訳プログラムは，認識プログラムのもう1つの構成例として役立つであろう．この認識プログラムは，さらに，その入力を翻訳する，あるいはより一般的には，その入力を処理するプログラムに拡張される．したがって，次のように議論を進めていこう．

ステップ 1. EBNF(拡張 BNF)と呼ばれるメタ言語の構文則を定義する．

ステップ 2. 5.4 節で与えられた規則によって，EBNF の認識プログラムを構成する．

ステップ 3. この認識プログラムを翻訳プログラムに拡張し，表制御型パー

ザと結合する．

このメタ言語（構文則生成言語）は，次の生成規則によって記述されるとする†．

$$\begin{aligned}
&\langle \text{production} \rangle &&::= \langle \text{symbol} \rangle = \langle \text{expression} \rangle. \\
&\langle \text{expession} \rangle &&::= \langle \text{term} \rangle \{ , \langle \text{term} \rangle \} \\
&\langle \text{term} \rangle &&::= \langle \text{factor} \rangle \{ \langle \text{factor} \rangle \} \\
&\langle \text{factor} \rangle &&::= \langle \text{symbol} \rangle \mid [\langle \text{term} \rangle]
\end{aligned} \tag{5.12}$$

生成規則入力言語のメタ記号を正確に表わすために，通常の BNF メタ記号とは異なるものが使われている．これには2つの理由がある．

1. (5.12) の中でメタ記号と言語記号とを区別する．
2. 計算機システムで普通利用されている文字を使う．特に ::= の代わりに，1文字の＝を使用できるようにする．

通常の BNF とわれわれの入力形との対応が**表 5.1** に示されている．さらに，各生成規則は，“．”によって明確に終わっている．この入力言語を使って例(5.7) を書くと次のようになる．

$$\begin{aligned}
&A = x, (B). \\
&B = AC. \\
&C = [+A].
\end{aligned} \tag{5.13}$$

構成すべき翻訳プログラムを簡単にするために，終端記号は1文字からなり，1行に1つの生成規則が書かれていると仮定する．また読みやすくするために，入力中に空白文字を挿入することを許すが，翻訳プログラムではこれを無視する．したがって，規則 B7 のステートメント *read* (*ch*) は，意味のある次の文字を得るためのルーチンの呼出しによって置き換えられなければならない．

表 5.1 メタ記号と言語記号

BNF	入力 *EBNF*
::=	=
|	,
{	[
}	]

† 訳注 production（生成規則），symbol（記号），expression（式），term（項），factor（因子）

これは，語い走査プログラム，あるいは単に**走査プログラム**と一般に呼ばれているものの非常に簡単な例である．走査プログラムの目的は，**文字**からなる入力列から次の**記号**——言語の表現規則から定められる——を抽出することである．これまでわれわれは記号と文字が同一であると考えてきたが，これは特殊な場合であり，実際にはこのようなことはまれである．

最後の規則として，入力 BNF では，非終端記号は文字 A から H によって，また終端記号は文字 I から Z によって表現されると仮定する．これは単に便宜上の規則であり，深い理由はない．これにより，生成規則を列挙するに先立って，終端記号や非終端記号を列挙する必要がなくなる．

(5.12) が制約規則 1 と 2 を満たすことを検証した後で，パーザ構成規則 B2 から B7 に厳密に従うことにより，(5.12) で規定される言語の認識アルゴリズムとして**プログラム 5.2** を得る．走査プログラムが *getsym* と呼ばれていることに注意せよ．

プログラム 5.2　言語(5.12)のパーザ

```
program parser(input, output);
label 99;
const empty = '*';
var sym: char;

procedure getsym;
begin
    repeat read(sym); write(sym) until sym ≠ ' '
end {getsym} ;

procedure error;
begin writeln;
    writeln (' INCORRECT INPUT'); goto 99
end {error} ;

procedure term;
    procedure factor;
    begin
        if sym in ['A' .. 'Z', empty] then getsym else
        if sym = '[' then
        begin getsym; term;
            if sym = ']' then getsym else error
        end else error
    end {factor} ;
```

```
begin factor;
    while sym in ['A' .. 'Z', '[', empty] do factor
end {term} ;

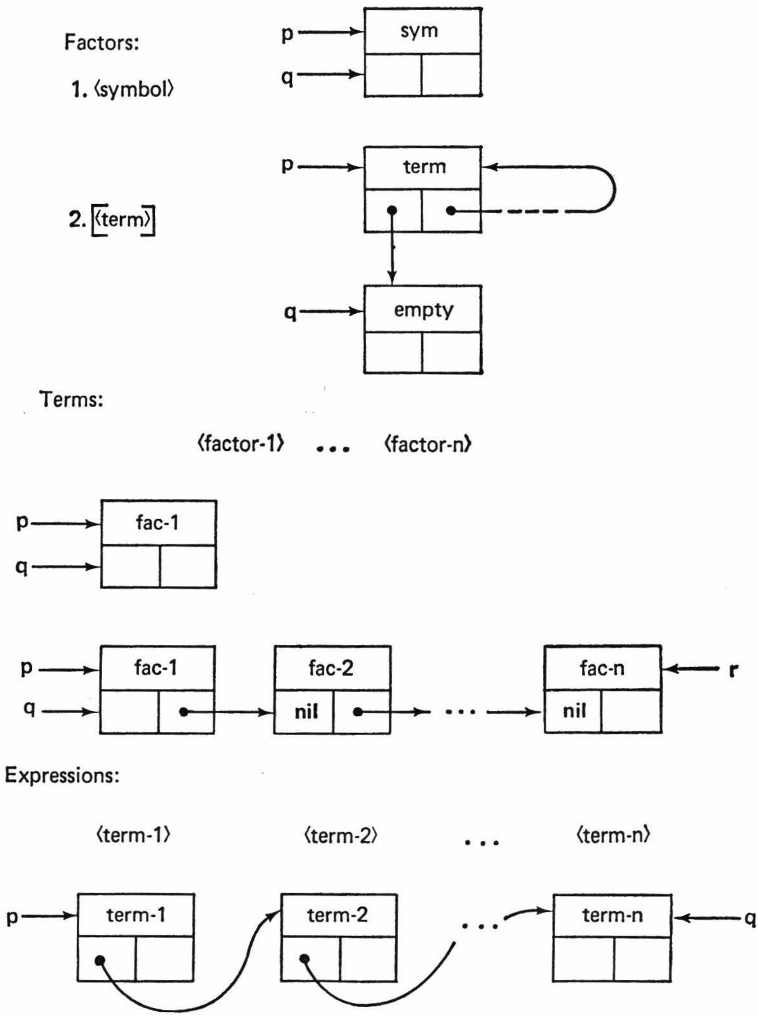
procedure expression;
begin term;
    while sym = ',' do
    begin getsym; term
    end
end {expression} ;

begin {main program}
    while ¬eof(input) do
    begin getsym;
        if sym in ['A' .. 'Z'] then getsym else error;
        if sym = '=' then getsym else error;
        expression;
        if sym ≠ '.' then error;
        writeln; readln;
    end ;
99: end
```

翻訳プログラム作成の第3ステップは，読み込まれたBNF生成規則を表現し，かつ，構文解析手続き(5.11)によって解釈されるデータ構造を構成することである．残念なことに，このステップは，認識アルゴリズムの構成に関するステップのようには定式化が可能でない．形式的な方法が存在しないので，各言語要素を表現するための構造を，再び図によって記述することにする．この結果得られる構造は，翻訳プログラムに拡張された対応する認識手続きの結果パラメータとして渡される．この結果としては，データ構造自身ではなくそれを指すポインタ p, q, r を返すのが自然である．(346頁の図参照)

明らかに，データ構造の新しい要素を生成するのは手続き *factor* の仕事である．残りの2つの手続きの仕事は，それらを連結し線形リストを作ることであるが，*term* は *suc* フィールドを，また *expression* は *alt* フィールドを連結のために用いる．その詳細はプログラム 5.3 から明らかである．

非終端記号の処理方法については，もう少し説明が必要である．非終端記号は，生成規則の左辺として現われる前に，⟨factor⟩ として現われることが可能である．非終端記号を表わすヘッダーのすべてを集めた線形リストの中に，記



号 *sym* が存在するかどうかを手続き *find* (*sym*, *h*) を使って調べる．もしその記号が見つかれば，*h* にそのポインタを代入する；もしこのリスト中にまだ存在しなければ，それをこのリストに加える．手続き *find* では，番人法を用いているが，これについては第4章で詳しく述べた．

プログラム5.3は3つの部分から成っているが，それらはそれぞれ入力デー

タの各部分に対応している．最初の部分では，生成規則を対応するデータ構造に変換する．２番目の部分では，その言語に属する文を生成する開始記号を読み込み同定する．(入力データの最初の部分と２番目の部分の間には，区切り記号が存在する)．第３番目の部分は構文解析プログラム (5.11) であり，最初の部分で生成されたデータ構造の制御のもとに入力文を読む．

プログラム 5.2 を変えることなく，単にステートメントを挿入しただけでプログラム 5.3 が得られたことは注目に値する．もとからあったプログラム 5.2 は，正しく構成された文の認識だけを行なうものであったが，プログラム 5.3 においては，認識だけでなく，受理された文の処理あるいは翻訳をも行なうようにプログラムが拡張されている．**段階的詳細化**，あるいはむしろ，**段階的機能増強**によって言語処理系を構成するこの方法は，大いに推奨できるものである．この方法により，設計者は，一時には，言語処理の限られた局面だけを扱うことができ，それによって，翻訳プログラムの正当性の検証作業が容易になるし，少なくともプログラム作成過程を通じて高い信頼度を保持することが可能になる．比較的簡単なこの例では，作成過程は２つの段階だけから構成されていたが，さらに複雑な言語の翻訳作業には，かなりの数の機能増強ステップが必要である．３つのステップからなる非常に似た作成過程が 5.8 節から 5.11 節で扱われるであろう．

プログラム 5.3 の作成過程から明らかなように，構文解析に対する構文表制御――あるいはデータ構造制御――アプローチには，個別パーザ・プログラム方式にない自由度と柔軟性がある．この柔軟性は一般には要求されないが，いわゆる**拡張可能言語**のためのコンパイラの必須条件である．拡張可能言語は，プログラマの裁量によってその構文要素を多少は拡張することが可能であるような言語である．プログラム 5.3 に対する入力と同様に，拡張可能言語のコンパイラに対する入力は，それ以後のプログラムで使用される言語拡張を指定する部分から成る．もっと野心的な方式では，翻訳すべきプログラムの部分と，新しい言語仕様の部分とを混在させることにより，コンパイルの過程で言語を変更することを許している．

これらの着想は魅力的であり，興味深いものではあるが，そのようなコンパイラを実現するための努力が報われたことはほとんどない．その理由は，構文や文認識の局面は翻訳作業全体の一部分に過ぎず，実際，副次的な部分でしかないからである．この部分はまた，最も形式化しやすい部分であり，したがって形式的な表の構造により最も容易に表現されるところである．これに反し，言語の**意味**，すなわち，翻訳の出力あるいは結果は，その形式化が非常に困難

な部分であり，ほぼ満足できるという形でさえ解決されていない問題である．これは，コンパイラ設計者が最初程には拡張可能言語に熱心でなくなった理由

プログラム 5.3　言語(5.12)の翻訳プログラム

```
program generalparser (input, output);
label 99;
const empty = '*';
type pointer = ↑node;
      hpointer = ↑header;
    node = record suc, alt: pointer;
                 case terminal: boolean of
                    true: (tsym: char);
                    false: (nsym: hpointer)
             end ;
    header = record sym: char;
                    entry: pointer;
                    suc: hpointer
               end ;
var list, sentinel, h: hpointer;
    p: pointer;
    sym: char;
    ok: boolean;

procedure getsym;
begin
    repeat read(sym); write(sym) until sym ≠ ' '
end {getsym} ;

procedure find(s: char; var h: hpointer);
{locate nonterminal symbol s in list. if not present, insert it}
    var h1: hpointer;
begin h1 := list; sentinel↑.sym := s;
    while h1↑.sym ≠ s do h1 := h1↑.suc;
    if h1 = sentinel then
        begin {insert} new (sentinel);
            h1↑.suc := sentinel; h1↑.entry := nil
        end ;
    h := h1
end {find} ;
```

```
procedure error;
begin writeln;
   writeln ('INCORRECT SYNTAX'); goto 99
end {error} ;

procedure term (var p,q,r: pointer);
   var a,b,c: pointer;
   procedure factor (var p,q: pointer);
      var a,b: pointer; h: hpointer;

   begin if sym in ['A' .. 'Z', empty] then
      begin {symbol}  new(a);
         if sym in ['A' .. 'H'] then
         begin {nonterminal} find(sym,h);
            a↑.terminal := false; a↑.nsym := h
         end else
         begin {terminal}
            a↑.terminal := true; a↑.tsym := sym
         end ;
         p := a; q := a; getsym
      end else
      if sym = '[' then
      begin getsym; term(p,a,b); b↑.suc := p;
         new(b); b↑.terminal := true; b↑.tsym := empty;
         a↑.alt := b; q := b;
         if sym = ']' then getsym else error
      end else error
   end {factor} ;

begin factor(p,a); q := a;
   while sym in ['A' .. 'Z', '[', empty] do
   begin factor(a↑.suc, b); b↑.alt := nil ; a := b
   end ;
   r := a
end {term} ;
```

```
procedure expression (var p,q: pointer);
   var a,b,c: pointer;
begin term(p,a,c): c↑.suc := nil;
   while sym = ',' do
   begin getsym;
      term(a↑.alt, b, c): c↑.suc := nil; a := b
   end ;
   q := a
end {expression} ;

procedure parse (goal: hpointer; var match: boolean);
   var s: pointer;
begin s := goal↑.entry;
   repeat
      if s↑.terminal then
      begin if s↑.tsym = sym then
                begin match := true; getsym
                end
              else match := (s↑.tsym = empty)
      end
      else parse(s↑.nsym, match);.
      if match then s := s↑.suc else s := s↑.alt
   until s = nil
end {parse} ;

begin {productions}
   getsym; new(sentinel); list := sentinel;
   while sym ≠ '$' do
   begin find(sym,h);
      getsym; if sym = '=' then getsym else error;
      expression (h↑.entry, p); p↑.alt := nil;
      if sym ≠ '.' then error;
      writeln; readln; getsym
   end ;
   h := list; ok := true;  {check whether all symbols are defined}
   while h ≠ sentinel do
   begin if h↑.entry = nil then
          begin writeln(' UNDEFINED SYMBOL ', h↑.sym);
                ok := false
          end ;
      h := h↑.suc
   end ;
```

```
    if ¬ok then goto 99;
{goal symbol}
    getsym; find(sym,h); readln; writeln;
{sentences}
    while ¬eof(input) do
        begin write(' '); getsym; parse(h,ok);
            if ok ∧ (sym='.') then writeln (' CORRECT')
                                else writeln (' INCORRECT');
            readln
        end ;
99: end .
```

を説明している．本章の残りの部分では，ひとつの小さな言語に対する簡単なコンパイラについて述べ，これによってわれわれの学習を終わることにする．

5.7 プログラム言語 PL/0

本章の残りの部分では，PL/0 と呼ばれる言語のコンパイラ作成について述べる．この言語は，そのコンパイラが本書の構成に適する程度にほどよく小さくなければならないという必要性と，それによって高級言語のコンパイラに関する基本概念がほとんど説明できるという要求を考慮して設計された．もちろん，もっと簡単な，あるいはもっと複雑な言語を選ぶこともできたであろう：PL/0 は，その説明を平易に行なえる程度に簡単であり，かつコンパイラ作成に価値がある程度に複雑である，というひとつの妥協案である．これよりかなり複雑な言語としては，PASCAL があるが，そのコンパイラも同じ技法を用いて構成された．PASCAL の構文則は補遺 II に与えられている．

プログラムの構造に関するかぎり，PL/0 は比較的完備している．ステートメントのレベルでの基本要素は，もちろん代入文である．構造化の概念には，順序づけ，条件実行および繰返しがあり，よく見なれた **begin/end**, **if** および **while** 文によって表現される．PL/0 にはサブルーチンの概念もあり，したがって，手続き宣言と手続き呼出しステートメントを含んでいる．

しかしながら，データ型に関しては簡単化の要求だけを考慮している．すなわち，整数だけがただ 1 つのデータ型である．この型の変数と定数とを宣言することが可能である．PL/0 にも通常の算術および比較演算子があることはもちろんである．手続き，すなわち，ある程度“閉じた”プログラムの部分が存在することにより，定数，変数，および手続きに**局所性**の概念を導入することが可能になる．したがって，PL/0 では各手続きの先頭で宣言を行ない，これ

らのものが，それが宣言されている手続きに対して局所的であることを指定する.

このような簡単な概説により，PL/0 の構文則は直感的に理解することができるが，その詳細は**図 5.4** に 7 つの図によって示されている．この図を等価な BNF 生成規則に変換する作業は，興味ある読者に任せる．プログラム言語全体の構文則が，図 5.4 のように簡潔で読みやすい形に定式化できることは，これらの図の表現力を示すものである.

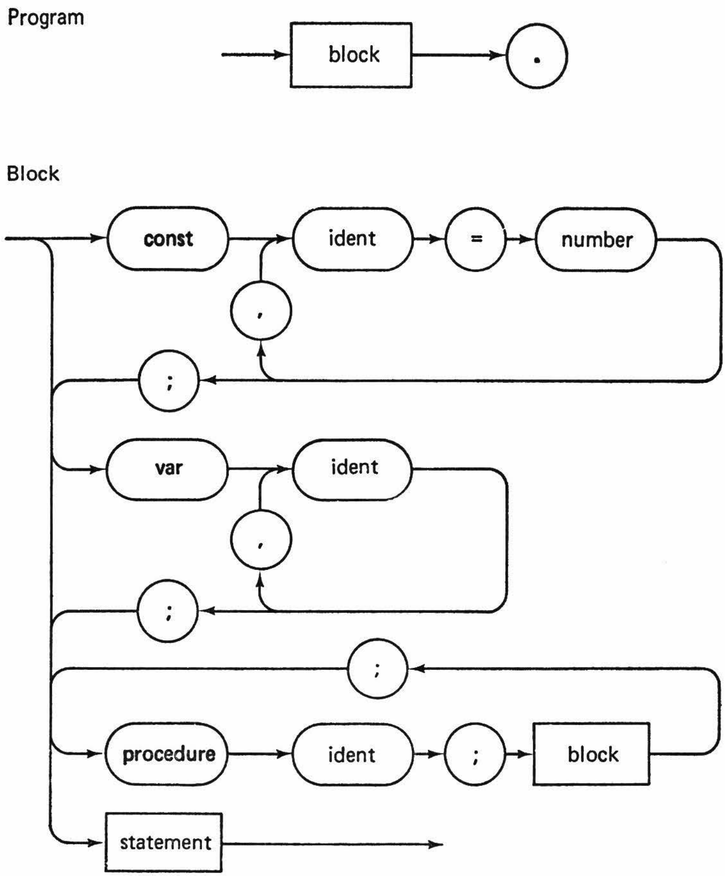


図 5.4　PL/0 の構文則

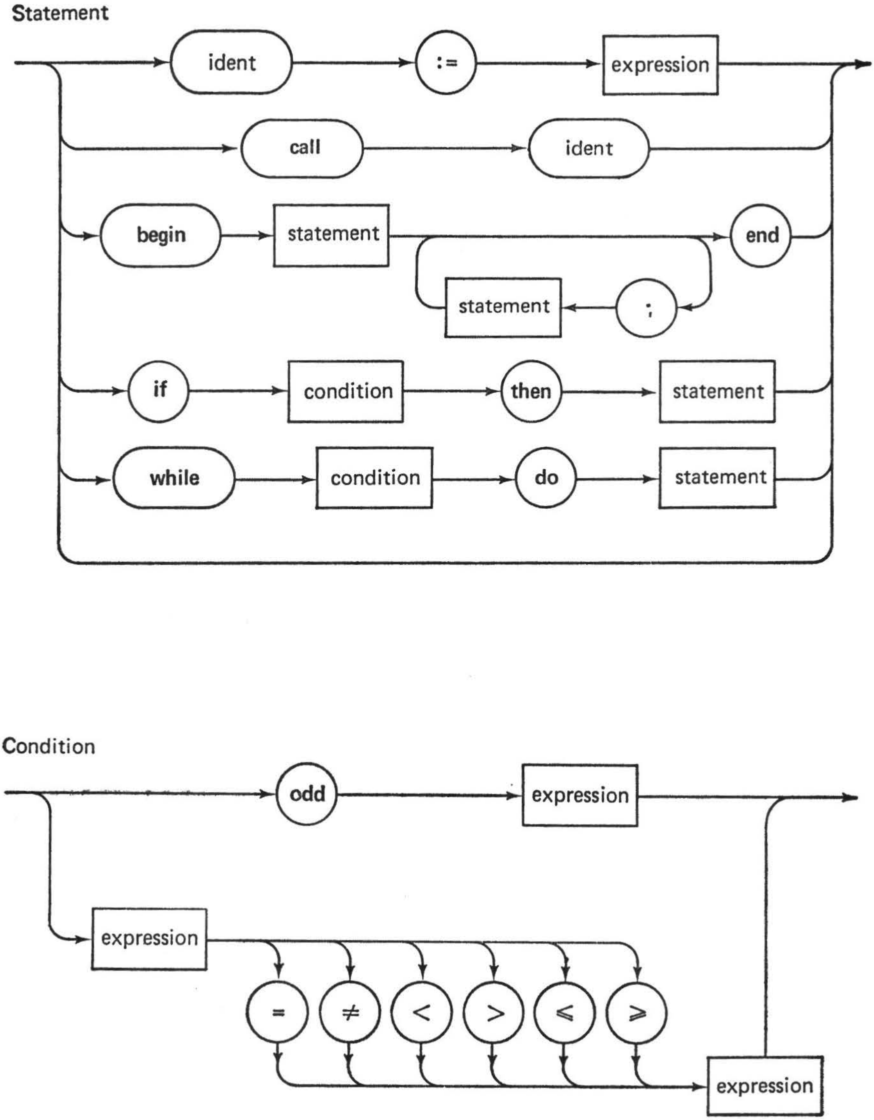
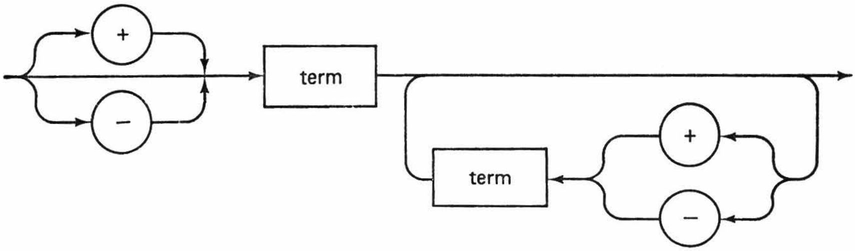
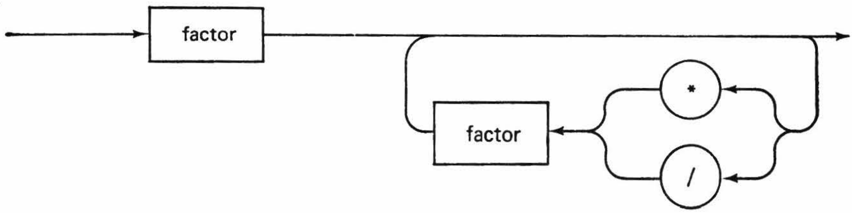


図5.4　PL/0 の構文則（続き）

Expression



Term



Factor

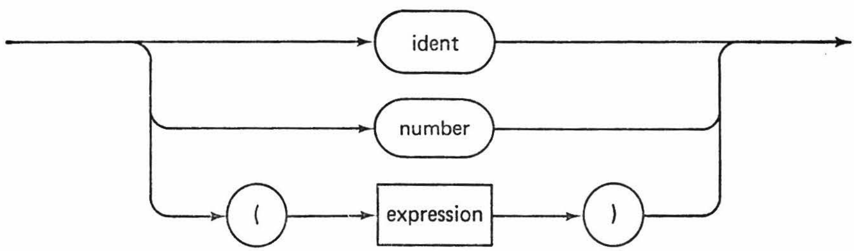


図5.4　PL/0の構文則（続き）

次のPL/0プログラムは，このミニ言語のもついくつかの機能の使い方を示している．このプログラムは，2つの自然数の乗算，除算および最大公約数(gcd)の計算を行なうよく知られたアルゴリズムを含んでいる．

```
const m = 7, n = 85;
var x,y,z,q,r;
procedure multiply;
    var a,b;
begin a := x; b := y; z := 0;                              (5.14)
        while b > 0 do
        begin
            if odd b then z := z + a;
            a := 2*a; b := b/2;
        end
end ;

procedure divide;
    var w;
begin r := x; q := 0; w := y;
    while w ≤ r do w := 2*w;
    while w > y do
        begin q := 2*q; w := w/2;
            if w ≤ r then
                begin r := r—w; q := q+1                   (5.15)
                end
        end
end ;

procedure gcd;
    var f,g;
begin f := x; g := y;
    while f ≠ g do
        begin if f < g then g := g—f;
                if g < f then f := f—g;
        end ;
    z := f
end ;                                                      (5.16)
begin
    x := m; y := n; call multiply;
    x := 25; y := 3; call divide;
    x := 84; y := 36; call gcd;
end
```

5.8 PL/0 のパーザ

PL/0 コンパイラへの第1ステップとして，パーザの構成を行なう．これは，5.7節で概説したパーザ構成規則 B1 から B7 に厳密に従って行なうことができる．しかしながら，この方法が適用できるためには，対象となる文法が制約規則1と2を満足していなければならない．したがって，構文グラフに適用できるように定式化されたこの規則の検証を行なわなければならない．

規則1は，ある分岐点から出ているすべての枝の最初の記号が異なっていなければならないことを指定している．図5.4の構文グラフについては，これは非常に容易に検証される．規則2は，いかなる記号も読まずに走査可能なグラフに適用される．PL/0 の構文則におけるそのようなグラフは，statement を記述しているものだけである．この場合，規則2は statement の後に続く最初の記号のすべてが，statement の先頭の記号と異なっていることを要求する．すべてのグラフに対して，その先頭の記号とそれに続く記号の集合を求めておくことが後で役に立つので，PL/0 構文則の7つの非終端記号(グラフ)すべて (program は除く) について，これらの集合を決定する．表5.2より，望んでいた結果，すなわち，statement の先頭の記号の集合と後続の記号の集合には共通部分のないことが確かめられる．したがって，パーザ構成規則 B1 から B7 を適用することができる．

表 5.2　PL/0 における先頭および後続記号

非終端記号 S	先頭記号 $L(S)$	後続記号 $F(S)$
block	**const var** **procedure** *ident* **if call begin while**	. ;
statement	*ident* **call** **begin if while**	. ; **end**
condition	**odd** + − (*ident number*	**then do**
expression	× − (*ident number*	. ;) R **end then do**
term	*ident number* (	. ;) R + − **end then do**
factor	*ident numder* (	. ;) R + − * / **end then do**

注意深い読者なら PL/0 の基本記号は，前の例のように1つの文字ではないことに気がついたであろう．その代わり，基本記号は**begin**や := のような文字の列である．プログラム5.3の場合と同様に，いわゆる走査プログラムは，入力記号列の表現あるいは語いに関する局面のみの処理を行なう．この走査プログラムは，次の記号を取り出すのがその仕事であるような手続き *getsym* と考えられ，その目的は次の通りである．

1. 区切り記号(空白文字)を読みとばす．
2. **begin, end** のような予約語を認識する．
3. 非予約語を識別名として認識する．その実際の識別名は，*id* と呼ばれる大域変数に代入される．
4. 数字の列を数として認識する．その実際の値は，大域変数 *num* に代入される．
5. := のような特殊文字の対を認識する．

入力文字列を処理するために，*getsym* は局所的な手続き *getch* を用いるが，この手続きは次の文字を取り出すためのものである．この主目的の他に，*getch* は次のようなことも行なう．

1. 行の終わりの情報を認識し，それを削除する．
2. 入力を出力ファイルにコピーし，プログラム・リストを生成する．
3. 各行の始めに，行番号あるいはロケーション・カウンタを印刷する．

走査プログラムでは，必要となる1記号の先読みを行なう．さらに，補助手続き *getch* も1文字の先読みを行なう．したがって，このコンパイラの先読みの合計は，1記号と1文字ということになる．

これらのルーチンの詳細は，プログラム5.4から明らかであるが，これには PL/0 のための完全なパーザが表現されている．実際には，このパーザは既に拡張されており，定数，変数あるいは手続きを表わす宣言された識別名を表に集める．そして，あるステートメント中で識別名が現われると，この表を探索し，その識別名が正しく宣言されていたか否かを決定する．もし正しく宣言されていなければ，当然，構文エラーと見なされる．なぜなら，それは“非合法”な記号を用いたという，プログラム・テキスト構成上の形式的誤りだからである．表に記憶された情報によってのみこの誤りが検出できるという事実は，この言語特有の**文脈依存性**の結果であるが，これはすべての識別名は適当な文脈において正しく宣言されなければならない，という規則によるものである．実際上すべてのプログラム言語は，この意味において文脈依存的である．それに

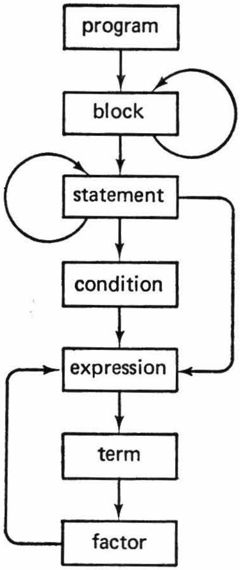


図 5.5　PL/0 の依存図

もかかわらず，文脈自由型の構文則はこれらの言語に対して非常に有用であり，その認識プログラムを系統的に構成する際に大いに役立つ．このようにして得られた枠組を拡張して，いくつかの文脈依存的要素を扱えるようにすることは非常に容易であり，これは現在のパーザへの識別名表の導入からもうかがうことができる．

個々の構文グラフに対応する個々のパーザ手続きを構成する前に，これらのグラフがどのように依存し合っているかを決定することは有用である．このために，いわゆる**依存図**を構成する．これは個々のグラフの関係を表示したものである．すなわち，任意のグラフ G に対して，G を定義するのに使われるすべてのグラフ $G_1, \cdots, G_n$ を列挙したものである．したがって，これは，1つの手続きから呼ばれる他の手続きを示している．PL/0 に対する依存図が図 **5.5** に示されている．

図 5.5 におけるループは，再帰の存在を示している．したがって，PL/0 コンパイラの実現に使われる言語には，再帰の禁止を課する必要がないことは本質的である．依存図からは，また，パーザ・プログラムの階層的構成に関する結果を導き出すことができる．たとえば，すべてのルーチンは構文要素 〈pro-

gram> を構文解析するルーチン（したがってこれは主プログラムである）に含まれる．（それに対して局所的であると宣言される）．さらに，<block> より下のすべてのルーチンは，目標 <block> の構文解析を表現しているルーチンに対して，局所的であると定義されるであろう．当然，これらすべてのルーチンは走査プログラム *getsym* を呼び出すが，これはまた *getch* を呼び出す．

プログラム 5.4 PL/0 パーザ

```
program PL0 (input, output);
{PL/0 compiler, syntax analysis only}
label 99;
const norw = 11;          {no. of reserved words}
   txmax = 100;           {length of indentifier table}
   nmax = 14;             {max. no of digits in numbers}
   al = 10;               {length of identifiers}
type symbol =
   (nul, ident, number, plus, minus, times, slash, oddsym,
    eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
    period, becomes, beginsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, constsym, varsym, procsym);
   alfa = packed array [1 .. al] of char;
   object = (constant, variable, procedure);
var ch: char;             {last character read}
   sym: symbol;           {last symbol read}
   id: alfa;              {last identifier read}
   num: integer;          {last number read}
   cc: integer;           {character count}
   ll: integer;           {line length}
   kk: integer;
   line: array [1 .. 81] of char;
   a: alfa;
   word: array [1 .. norw] of alfa;
   wsym: array [1 .. norw] of symbol;
   ssym: array [char] of symbol;
   table: array [0 .. txmax] of
          record name: alfa;
                 kind: object
          end ;
```

```
procedure error (n: integer);
begin writeln (' ':cc, '↑', n:2); goto 99
end {error} ;
procedure getsym;
   var i,j,k: integer;
   procedure getch;
   begin if cc = ll then
       begin if eof(input) then
                begin write (' PROGRAM INCOMPLETE'); goto 99
                end ;
          ll := 0; cc := 0; write(' ');
          while ¬eoln(input) do
             begin ll := ll+1; read(ch); write(ch); line[ll] := ch
             end ;
          writeln; ll := ll+1; read(line[ll])
       end ;
       cc := cc+1; ch := line[cc]
   end {getch} ;
begin {getsym}
while ch = ' ' do getch;
   if ch in ['A' .. 'Z'] then
   begin {identifier or reserved word}  k := 0;
      repeat if k < al then
         begin k := k+1; a[k] := ch
         end ;
         getch
      until ¬(ch in ['A' .. 'Z', '0' .. '9']);
      if k ≥ kk then kk := k else
         repeat a[kk] := ' '; kk := kk-1
         until kk = k;
      id := a; i := 1; j := norw;
      repeat k := (i+j) div 2;
         if id ≤ word[k] then j := k-1;
         if id ≥ word[k] then i := k+1
      until i > j;
      if i-1 > j then sym := wsym[k] else sym := ident
   end else
```

```
    if ch in ['0' .. '9'] then
    begin {number} k := 0; num := 0; sym := number;
       repeat num := 10*num + (ord(ch)-ord('0'));
          k := k+1; getch
       until ¬(ch in ['0' .. '9']);
       if k > nmax then error (30)
    end else
    if ch = ':' then
    begin getch;
       if ch = '=' then
       begin sym := becomes; getch
       end else sym := nul;
    end else
    begin sym := ssym[ch]; getch
    end
end {getsym} ;

procedure block (tx: integer);
    procedure enter (k: object);
    begin {enter object into table}
       tx := tx + 1;
       with table[tx] do
       begin name := id; kind := k;
       end
    end {enter} ;

    function position (id: alfa): integer;
       var i: integer;
    begin {find identifier id in table}
       table[0].name := id; i := tx;
       while table[i].name ≠ id do i := i-1;
       position := i
    end {position} ;
```

```
    procedure constdeclaration;
    begin if sym = ident then
        begin getsym;
            if sym = eql then
            begin getsym;
                if sym = number then
                    begin enter (constant); getsym
                    end
                else error (2)
            end else error (3)
        end else error (4)
    end {constdeclaration} ;

    procedure vardeclaration;
    begin if sym = ident then
            begin enter (variable); getsym
            end else error (4)
    end {vardeclaration} ;

procedure statement;
    var i: integer;
    procedure expression;
        procedure term;
            procedure factor;
                var i: integer;
            begin
                if sym = ident then
                begin i := position(id);
                    if i = 0 then error (11) else
                    if table[i] .kind = procedure then error (21);
                    getsym
                end else
                if sym = number then
                begin getsym
                end else
                if sym = lparen then
                begin getsym; expression;
                    if sym = rparen then getsym else error (22)
                end
                else error (23)
            end {factor} ;
```

```
    begin {term} factor;
        while sym in [times, slash] do
            begin getsym; factor
            end
    end {term} ;
begin {expression}
    if sym in [plus, minus] then
        begin getsym; term
        end else term;
    while sym in [plus, minus] do
        begin getsym; term
        end
end {expression} ;
procedure condition;
begin
    if sym = oddsym then
    begin getsym; expression
    end else
    begin expression;
        if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
            error (20) else
        begin getsym; expression
        end
    end
end {condition} ;
begin {statement}
if sym = ident then
begin i := position(id);
    if i = 0 then error (11) else
    if table [i] .kind ≠ variable then error (12);
    getsym; if sym = becomes then getsym else error (13);
    expression
end else
if sym = callsym then
```

```
        begin getsym;
            if sym ≠ ident then error (14) else
                begin i := position(id);
                    if i = 0 then error (11) else
                    if table[i] .kind ≠ procedure then error (15);
                    getsym
                end
        end else
        if sym = ifsym then
        begin getsym; condition;
            if sym = thensym then getsym else error (16);
            statement;
        end else
        if sym = beginsym then
        begin getsym; statement;
            while sym = semicolon do
                begin getsym; statement
                end ;
            if sym = endsym then getsym else error (17)
        end else
        if sym = whilesym then
        begin getsym; condition;
            if sym = dosym then getsym else error (18);
            statement
        end
    end {statement} ;
begin {block}
    if sym = constsym then
    begin getsym; constdeclaration;
        while sym = comma do
            begin getsym; constdeclaration
            end ;
        if sym = semicolon then getsym else error (5)
    end ;
    if sym = varsym then
    begin getsym; vardeclaration;
        while sym = comma do
            begin getsym; vardeclaration
            end ;
        if sym = semicolon then getsym else error (5)
    end ;
```

```
   while sym = procsym do
   begin getsym;
      if sym = ident then
         begin enter (procedure); getsym
         end
      else error (4);
      if sym = semicolon then getsym else error (5);
      block (tx);
      if sym = semicolon then getsym else error (5);
   end ;
   statement
end {block} ;
begin {main program}
   for ch := 'A' to ';' do ssym[ch] := nul;
   word[ 1] := 'BEGIN ';     word[ 2] := 'CALL      ';
   word[ 3] := 'CONST';      word[ 4] := 'DO        ';
   word[ 5] := 'END   ';     word[ 6] := 'IF        ';
   word[ 7] := 'ODD   ';     word[ 8] := 'PROCEDURE';
   word[ 9] := 'THEN  ';     word[10] := 'VAR       ';
   word[11] := 'WHILE ';
   wsym[ 1] := beginsym;     wsym[ 2] := callsym;
   wsym[ 3] := constsym;     wsym[ 4] := dosym;
   wsym[ 5] := endsym;       wsym[ 6] := ifsym;
   wsym[ 7] := oddsym;       wsym[ 8] := procsym;
   wsym[ 9] := thensym;      wsym[10] := varsym;
   wsym[11] := whilesym;
   ssym['+'] := plus;        ssym['-'] := minus;
   ssym['*'] := times;       ssym['/'] := slash;
   ssym['('] := lparen;      ssym[')'] := rparen;
   ssym['='] := eql;         ssym[','] := comma;
   ssym['.'] := period;      ssym['≠'] := neq;
   ssym['<'] := lss;         ssym['>'] := gtr;
   ssym['≤'] := leq;         ssym['≥'] := geq;
   ssym[';'] := semicolon;
   page(output);
   cc := 0; ll := 0; ch := ' '; kk := al; getsym;
   block (0);
   if sym ≠ period then error (9);
99: writeln
end
```

5.9 構文エラーからの回復

これまでパーザは，入力記号列がある言語に属するか否かを決定する，という控え目な動作だけを行なってきた．またその副産物として，文のもつ固有な構造を求めることもパーザは行なってきた．しかしながら，正しく構成されてない構文に出合うと，パーザはただちにその作業をやめ，したがってプログラムもまた停止することが許された．もちろん，実用的なコンパイラに対しては，このような方式をとることはできない．その代わりに，コンパイラは適当なエラーの診断を行ない，――さらにエラーを見つけるために――構文解析を続行することができなければならない．この続行が可能となるためには，エラーの性質や正しくないプログラムを書いた人の意図について，何らかの仮定を置くか，あるいは入力列中のそれ以後の適当な部分を読みとばすか，またはその両方を行なわなければならない．正しさの度合いの高い仮定を選ぶ技術は，どちらかと言えば複雑であり，どのような形のものであれ，今までその形式化に成功したことはない．その理由は，構文則や構文解析の形式化においては，人間の精神に影響を与える多くの要因が考慮されていないからである．たとえば，セミコロンのような句読点を忘れるのは，よくある誤りであるが（プログラムだけではない！），算術式においてプラス（+）演算子を書き忘れることはほとんどありえない．パーザにとっては，セミコロンもプラスも単なる終端記号であり，それ以上の区別は行なわない．一方，人間にとっては，セミコロンはほとんど意味を持たず，行の終わりにある冗長なものでしかないが，算術演算子の重要性は疑う余地のないものである．十分な回復システムを設計する場合に考慮すべきそのような事柄は，他にももっとたくさんあるが，すべての文脈自由言語の枠組の中でそれを一般化することは不可能である．

それにもかかわらず，PL/0のような特定な言語以外のものについても成立すると考えられ，また，有効であるような，規則やヒントがいくつかある．それら特徴は，パーザの回復メカニズムだけではなく，言語の設計方針にも関連している．まず最初に，**簡単な言語構造**だけを用いても，意味のある回復が大いに行ないやすくなったり，さらには可能になることさえあるということは明らかである．特に，エラーの診断を行なって，後に続く入力列のある部分を読みとばす(無視する)場合には，誤ることがほとんどないような**キー語**がこの言語に含まれており，これによってパーザを正常な状態に戻すことができることが絶対に必要である．PL/0は特にこの規則に従っている：すべての構造をもった文は，**begin**, **if**, **while** のような誤ることのないキー語で始まっている．

また，宣言についても同様で，**var**，**const**，**procedure** がその先頭についている．この規則を**キー語規則**と呼ぶことにする．

次の規則は，パーザの構成ともっと直接的に関係している。トップダウン・パーザの特徴は，目標を部分目標に分割し，その部分目標を処理するために他のパーザを呼び出すことである．パーザがエラーを発見した場合，単に先に進むのを拒み親のパーザにエラーの出現を報告するだけではいけないということを，この規則は規定している．その代わりに，パーザは，もっともらしい解析を再開できる点までテキストを自ら読み進まなければならない．したがって，われわれはこの規則を，**あわてない規則**(don't panic rule)とよぶことにする．この規則のプログラム上の帰結は，正常な停止点以外にはパーザからの出口がないということである．

あわてない規則の厳密な1つの解釈は，非合法な構成に出合ったときに，現在解析中の構文要素の次に現われる可能性のある記号まで，入力列を読みとばすことである．このためには，すべてのパーザが，現在の呼出し点における後続記号の集合を知っていなければならない．

したがって，最初の詳細化(または機能増強)ステップでは，各構文解析手続きに，可能な後続記号の集合を表わす陽なパラメータ *fsys* を与えるべきである．そして，各手続きの終わりに，入力テキストの次の記号が本当にこれらの後続記号の中にあることを検証するためのテストを挿入する（この条件がプログラム中の論理によって既に検証されていなければ)．

しかしながら，すべての場合について，そのような後続記号の次の出現まで入力テキストを読みとばすことは，非常に近視眼的である．なぜなら，プログラマがちょうど1つの記号（たとえばセミコロン）だけを誤って書き忘れたかも知れないからである．この場合，次の後続記号までのテキスト全体を無視しては被害が大きすぎる．したがって，読みとばしを停止させる記号の集合に，見落とすべきではない構文の先頭を特に示しているキー語を追加する．ゆえに，構文解析プログラムにパラメータとして渡される記号は，単に後続記号だけではなく，**停止記号**も含まれる．停止記号の集合は，別個のキー記号によって初期化され，構文解析の部分目標の階層に入っていくにしたがって，徐々に正しい後続記号によって増補される，と考えることができる．柔軟性のために，*test* と呼ばれる一般的なルーチンを導入し，上で述べた検証を行なう．この手続き(5.17)は3つのパラメータをもっている：

1. 許容される次の記号の集合 *s1*；もし現在の記号がこの中になければ，エラーが起こっている．

2.　それが現われることは確かにエラーではあるが，どのような場合でも無視や読みとばしを行なってはならない付加的な停止記号の集合 $s2$.

3.　適切なエラー診断情報の番号 n.

```
procedure test (s1, s2: symset; n: integer);
begin if ¬(sym in s1) then
      begin error(n); s1 := s1+s2;                    (5.17)
            while ¬(sym in s1) do getsym
      end
end
```

手続き(5.17)は，構文解析手続きの入口において，現在の記号が最初の記号として許されるかどうかを検証するために使っても便利であろう．これは，ステートメント

if $sym = a_1$ **then** S_1 **else**

$\cdots\cdots$

if $sym = a_n$ **then** S_n **else** X

のように，構文解析手続き X が無条件によばれる場合には，常に推奨される方法である．なおこのステートメントは，生成規則

$$A ::= a_1 S_1 | \cdots | a_n S_n | X \tag{5.18}$$

を翻訳した結果である．この例については，パラメータ $s1$ は X の先頭記号集合でなければならず，一方，$s2$ としては A の後続記号集合が選ばれる（表5.2参照）．この手続きの詳細はプログラム 5.5 に与えられているが，これはプログラム 5.4 の機能増強版である．読者の便宜のために，パーザ全体を再び載せてあるが，大域変数の初期化と手続き *getsym* については変わっていないので省略した．

これまで提案してきた方式は，入力テキスト中の1個以上の記号を無視することによって回復を行ない，正常状態に戻そうとする性質を持っていた．しかし，記号の**省略**によってエラーが生じた場合には，この方式は適当ではない．経験によればこのようなエラーは，構文的な機能は持っているが動作を表現していないような記号に実質的には限定される．PL/0 におけるセミコロンは，そのような例である．この場合，もし後続記号集合が適当なキー語によって増補されていれば，パーザが早計な読みとばしを行なうのを止めさせ，これによ

って，あたかも省略された記号が挿入されたかのように振舞わせることができる．これは，複合文を解析するプログラムの一部分である(5.19)からもわかる．それは省略されたセミコロンをキー語の前に実質的に“挿入”する．*statbegsys* と呼ばれる集合は，構文 statement の先頭記号の集合である．

```
if sym = beginsym then
begin getsym;
    statement([semicolon, endsym]+fsys);
    while sym in [semicolon]+statbegsys do
        begin
            if sym = semicolon then getsym else error;        (5.19)
            statement([semicolon, endsym]+fsys)
        end;
    if sym = endsym then getsym else error
end
```

このプログラムが構文エラーを診断し，異常な状況から回復するのに成功する度合いは，PL/0 プログラム (5.20) を考えることにより推定することができる．このリストは，プログラム 5.5 によって作られた出力を表わしており，また，**表 5.3** はプログラム 5.5 中のエラー番号に対応する診断メッセージの集合である．

次のプログラム (5.20) は，(5.14) から (5.16) に構文エラーを加えることにより得られたものである．

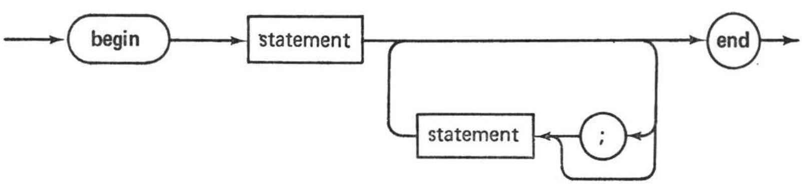


図 5.6 修正された複合文の構文則

表5.3　PL/0 コンパイラのエラー・メッセージ

1.	:= の代わりに = を用いよ.
2.	= の後に数がこなければならない.
3.	= の後に識別名がこなければならない.
4.	**const, var, procedure** の後に識別名がこなければならない.
5.	セミコロンかコンマがない.
6.	手続き宣言の後に不正な記号がある.
7.	ステートメントがあるはずである.
8.	ブロック中のステートメントの次に不正な記号がある.
9.	ピリオドがあるはずである.
10.	ステートメントの間のセミコロンがぬけている.
11.	識別名が宣言されていない.
12.	定数または手続きへの代入は許されない.
13.	代入演算子 := があるはずである.
14.	**call** の後に識別名が必要である.
15.	定数あるいは変数の呼出しは意味がない.
16.	**then** があるはずである.
17.	セミコロンまたは **end** があるはずである.
18.	**do** があるはずである.
19.	ステートメントの後に不正な記号がある.
20.	関係演算子があるはずである.
21.	式に手続きが含まれてはいけない.
22.	右括弧がぬけている.
23.	直前の因子の次にこの記号が続くことはできない.
24.	式はこの記号で始まってはいけない.
30.	この数は大きすぎる.

```
const m = 7, n = 85
var x,y,z,q,r;
   ↑ 5
          ↑ 5
procedure multiply;
    var a,b                                          (5.20)
    begin a := u; b := y; z := 0
    ↑ 5
          ↑11
        while b > 0 do
          ↑10
```

```
        begin
           if odd b do z := z + a,
                    ↑16
                    ↑19
           a := 2a; b := b/2;
                 ↑23
        end
     end ;
procedure divide
     var w;
        ↑ 5
     const two = 2, three := 3;
        ↑ 7
                        ↑ 1
begin r = x; q := 0; w := y;
         ↑13
         ↑24
     while w ≤ r do w := two*w;
     while w > y                                  (5.20)
        begin q := (2*q; w := w/2);
           ↑18
                    ↑22
                             ↑23
           if w ≤ r then
              begin r := r−w q := q+1
                          ↑23
              end
        end
end ;
procedure gcd;
     var f,g;
begin f := x; g := y
     while f ≠ g do
        ↑17
        begin if f < g then g := g−f;
              if g < f then f := f−g;
     z := f
end ;
```

```
begin
    x := m; y := n; call multiply;
    x := 25; y := 3; call divide;
    x := 84; y := 36; call gcd;
    call x; x := gcd; gcd =x
        ↑15
                    ↑21
                            ↑12                                    (5.20)
                              ↑13
                              ↑24
end
     ↑17
     ↑ 5
     ↑ 7
PROGRAM INCOMPLETE
```

正しい文を程よく能率的に翻訳する方式では，可能な不正構文のすべてを合理的な仕方で処理することは，明らかに不可能である．これはあたりまえである！　普通の努力によって作られたどんな方式にもうまくゆかない場合がある．すなわち，不十分にしか処理できない不正構文があるであろう．しかしながら，よいコンパイラは次のような重要な特性をもっていなければならない．

1. どのような入力に対しても，コンパイラは総くずれになってはいけない，
2. 言語の定義に対して非合法な構文すべてを検出し，印をつける．
3. それなりに頻繁に起こり，本当にプログラマの誤り（見落しや誤解による）であるようなエラーは正しく診断し，それ以外のいかなる“よろめき”(いわゆる“にせ”のエラー・メッセージ)も起こさない．

改良の余地はあるが，ここで述べた方式は満足に動作する．その長所は，いくつかの基本的規則に従って系統的に構成されていることである．これらの基本的規則に付加すべきものは，発見的手法あるいは言語の実際の使用経験に基づいて決定されるパラメータだけである．

プログラム5.5 エラー回復を行なう PL/0 パーザ

```
program PL0 (input ,output);
{PL/0 compiler with syntax error recovery}
label 99;
const norw = 11;        {no. of reserved words}
   txmax = 100;         {length of identifier table}
   nmax = 14;           {max. no. of digits in numbers}
   al = 10;             {length of identifiers}
type symbol =
   (nul, ident, number, plus, minus, times, slash, oddsym,
   eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
   period, becomes, beginsym, endsym, ifsym, thensym,
   whilesym, dosym, callsym, constsym, varsym, procsym);
   alfa = packed array [1 .. al] of char;
   object = (constant, variable, procedure);
   symset = set of symbol;
var ch: char;           {last character read}
   sym: symbol;         {last symbol read}
   id: alfa;            {last identifier read}
   num: integer;        {last number read}
   cc: integer;         {character count}
   ll: integer;         {line length}
   kk: integer;
   line: array [1 .. 81] of char;
   a: alfa;
   word: array [1 .. norw] of alfa;
   wsym: array [1 .. norw] of symbol;
   ssym: array [char] of symbol;
   declbegsys, statbegsys, facbegsys: symset;
   table: array [0 .. txmax] of
         record name: alfa;
               kind: object
         end ;
procedure error (n: integer);
begin writeln(' ':cc, '↑', n: 2);
end {error} ;
```

```
procedure test (s1,s2: symset; n: integer);
begin if ¬(sym in s1) then
        begin error(n); s1 := s1 + s2;
            while ¬(sym in s1) do getsym
        end
end {test} ;
procedure block (tx: integer; fsys: symset);
    procedure enter (k: object);
    begin {enter object into table}
        tx := tx + 1;
        with table[tx] do
        begin name := id; kind := k;
        end
    end {enter} ;
    function position (id: alfa): integer;
        var i: integer;
    begin {find identifier id in table}
        table[0] .name := id; i := tx;
        while table[i] .name ≠ id do i := i−1;
        position := i
    end {position} ;
    procedure constdeclaration;
    begin if sym = ident then
        begin getsym;
            if sym in [eql, becomes] then
            begin if sym = becomes then error (1);
                getsym;
                if sym = number then
                    begin enter (constant); getsym
                    end
                else error (2)
            end else error (3)
        end else error (4)
        end {constdeclaration} ;
    procedure vardeclaration;
    begin if sym = ident then
            begin enter (variable); getsym
            end else error (4)
    end {vardeclaration} ;
```

```
procedure statement (fsys: symset);
var i: integer;
procedure expression (fsys: symset);
    procedure term (fsys: symset);
        procedure factor (fsys: symset);
            var i: integer;
        begin test (facbegsys, fsys, 24);
            while sym in facbegsys do
            begin
                if sym = ident then
                begin i := position (id);
                    if i = 0 then error (11) else
                    if table[i] .kind = procedure then error (21);
                    getsym
                end else
                if sym = number then
                begin getsym;
                end else
                if sym = lparen then
                begin getsym; expression ([rparen]+fsys);
                    if sym = rparen then getsym else error (22)
                end ;
                test(fsys, [lparen], 23)
            end
        end {factor} ;
    begin {term} factor (fsys+[times, slash]);
        while sym in [times, slash] do
            begin getsym; factor(fsys+[times, slash])
            end
    end {term} ;
begin {expression}
    if sym in [plus ,minus] then
        begin getsym; term(fsys+[plus, minus])
        end else term(fsys+[plus, minus]);
    while sym in [plus, minus] do
        begin getsym; term(fsys+[plus, minus])
        end
end {expression} ;
```

```
procedure condition(fsys: symset);
begin
   if sym = oddsym then
   begin getsym; expression(fsys);
   end else
   begin expression ([eql, neq, lss, gtr, leq, geq]+fsys);
      if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
         error (20) else
      begin getsym; expression (fsys)
      end
   end
end {condition} ;
begin {statement}
   if sym = ident then
   begin i := position(id);
      if i = 0 then error (11) else
      if table[i] .kind ≠ variable then error (12);
       getsym; if sym = becomes then getsym else error (13);
      expression(fsys);
   end else
   if sym = callsym then
   begin getsym;
      if sym ≠ ident then error (14) else
         begin i := position(id);
            if i = 0 then error (11) else
            if table[i] .kind ≠ procedure then error (15);
            getsym
         end
   end else
   if sym = ifsym then
   begin getsym; condition ([thensym, dosym]+fsys);
      if sym = thensym then getsym else error (16);
      statement(fsys)
   end else
```

```
    if sym = beginsym then
    begin getsym; statement([semicolon, endsym]+fsys);
        while sym in [semicolon]+statbegsys do
        begin
            if sym = semicolon then getsym else error (10);
            statement([semicolon ,endsym]+fsys)
        end ;
        if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin getsym; condition([dosym]+fsys);
        if sym = dosym then getsym else error (18);
        statement(fsys);
    end ;
    test(fsys, [ ], 19)
end {statement} ;
begin {block}
    repeat
        if sym = constsym then
        begin getsym;
            repeat constdeclaration;
                while sym = comma do
                    begin getsym; constdeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident
        end ;
        if sym = varsym then
        begin getsym;
            repeat vardeclaration;
                while sym = comma do
                    begin getsym; vardeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident;
        end ;
        while sym = procsym do
        begin getsym;
            if sym = ident then
                begin enter (procedure); getsym
                end
            else error (4);
```

```
            if sym = semicolon then getsym else error (5);
            block (tx, [semicolon]+fsys);
            if sym = semicolon then
               begin getsym; test(statbegsys+[ident, procsym], fsys, 6)
               end
            else error (5)
         end ;
         test(statbegsys+[ident], declbegsys, 7)
      until ¬(sym in declbegsys);
      statement([semicolon, endsym]+fsys);
      test(fsys, [ ], 8);
   end {block} ;
   begin {main program}
      . . . Initialization (see Program 5.4) . . .
      cc := 0; ll := 0; ch := ' '; kk := al; getsym;
      block (0, [period]+declbegsys+statbegsys);
      if sym ≠ period then error (9);
   99: writeln
   end
```

5.10 PL/0計算機

これまでは，コード生成の行なわれる計算機についての知識を全く用いずに，PL/0コンパイラの作成を行なってきたが，これは非常に注目すべきことである．対象となる計算機の構造が，コンパイラの構文解析やエラー回復方式に影響を与えるはずはないし，またそうであってはならない．その代わり，どのような計算機に対するものであれ，適切なコード生成方式であれば，段階的詳細化法によって既存のパーザの上に重畳できるものでなければならない．以下ではこれを行なうが，そのために，コンパイルの対象となる計算機を選択することが必要となる．

コンパイラの記述が程よく簡単になり，また現実の計算機の特殊な性質を特に考慮しないで済むように，PL/0の要求に合わせた，われわれに都合のよい計算機を仮定しよう．この計算機はハードウェアとしては現実には存在しないので，仮想的な計算機である；われわれはこれを**PL/0計算機**とよぶことにする．

本節の目的は，この形の計算機アーキテクチャが選ばれるに到った理由を説明することではない．むしろ，その直感的な紹介を行ない，この計算機の詳細

な定義をアルゴリズムによって記述したマニュアルとすることである．この定式化は，アルゴリズムによる実際の計算機の正確かつ詳細な記述例として役立つであろう．このアルゴリズムは，PL/0 計算機の命令を逐次解釈するものであり，**解釈プログラム(インタープリタ)**と呼ばれる．

PL/0 計算機は 2 つの記憶部，命令レジスタおよび 3 つの番地レジスタからなる．**コード**と呼ばれる**プログラム記憶部**はコンパイラによってロードされ，そのコードの解釈中不変である．したがって，それは読出し専用記憶装置と考えることができる．**データ記憶部** S は**スタック**として構成されており，すべての算術演算はスタックの先頭の 2 つの要素に作用し，それらをその実行結果によって置き換える．先頭の要素は，**スタック・トップ・レジスタ** T により指定(インデックス)される．**命令レジスタ** I は，現在解釈中の命令を保持する．**プログラム番地レジスタ** P は，次に解釈されるべき命令を指示する．

PL/0 のどの手続きも局所変数を含んでいるであろう．手続きは再帰的に起動され得るので，これらの局所変数に対する記憶領域は，実際の手続き呼出し以前に割り付けることは不可能である．したがって，各手続きに対するデータ領域は，スタック記憶部 S に連続的に積み上げられる．手続きの起動は厳密に先入れ後出し方式に従うので，スタックは記憶割付け方式として適当なものである．各手続きはそれ個有のいくらかの内部情報，すなわち，その呼出しのプログラム番地（いわゆる**戻り番地**）と呼出しプログラムのデータ領域の番地を持っている．これらの 2 つの番地は，手続きの終了後にプログラムの再開を正しく行なうために必要なものである．これらは，内部変数あるいは暗黙の局所変数と考えられ，その手続きのデータ領域に割り付けられる．われわれは，これらを**戻り番地** RA と**動的リンク** DL と呼ぶ．動的リンクの始点，すなわち，最も最近割り付けられたデータ領域の番地は，**ベース番地レジスタ** B に保持される．

記憶領域の実際の割付けは実行(解釈)時に行なわれるので，コンパイラは絶対番地を含んだ命令を生成することはできない．コンパイラのできることは，あるデータ領域内での変数の位置（変位）を決定することだけであるので，**相対番地**しか与えることができない．したがって，解釈プログラムは，この**変位**を適当なデータ領域のベース番地に加えなければならない．もし，ある変数が現在解釈中の手続きに対して局所的であるならば，このベース番地は B レジスタによって与えられる．もしそうでなければ，それはデータ領域のチェインをたどってゆくことにより求めなければならない．しかしながら，コンパイラが知っているのはアクセス・パスの静的な深さだけであり，これに対し動的リン

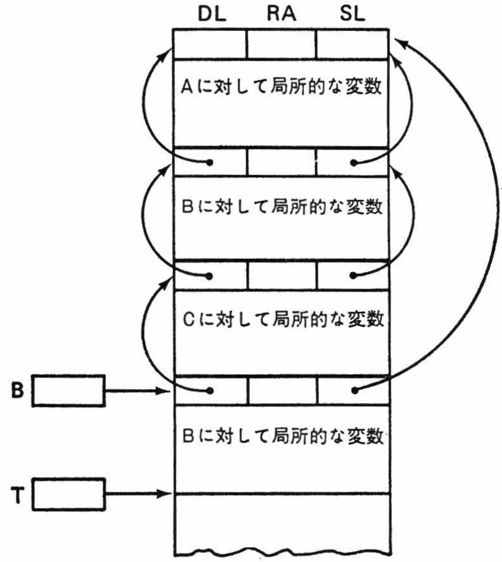


図5.7　PL/0 計算機のスタック

クのチェインは，手続き呼出しの動的な履歴を保存したものである．残念なことに，これらの2つのパスは必ずしも同一とは限らない．

たとえば，手続き A が，A に対して局所的と宣言された手続き B を呼び，B が，B に対して局所的と宣言された手続き C を呼び，さらに C が B を（再帰的に）呼んだと仮定する．このとき，A はレベル1で，B はレベル2で，また C はレベル3で宣言されたという（**図5.7**参照）．もし A の中で宣言された変数 a が B の中でアクセスされたとすると，コンパイラは B と A の間に**レベル差**が1あることを知る．しかし，動的リンクのチェインを1ステップをたどると，C に対して局所的な変数が得られてしまうであろう．

したがって，2番目のリンクによるチェインを設けて，コンパイラがこの状況を見ることができるように，データ領域をリンクしなければならない．われわれはこれを静的リンク SL と呼ぶ．

したがって，番地は，静的レベル差を表わす数とデータ領域内における相対変位の対として生成される．われわれは，データ記憶部の各場所には番地または整数が格納できると仮定する．

PL/0 計算機の命令集合は，PL/0 言語の要求に合わせて定められ，次のような命令を含んでいる．

1. スタックの先頭に数(リテラリ)をロードするための命令(LIT)
2. スタックの先頭に変数の値を取り出すための命令(LOD)
3. 代入文に対応するストア命令(STO)
4. 手続き呼出しに対応する，サブルーチンを起動するための命令(CAL)
5. スタック・ポインタ T を増加させることにより，スタック上に記憶領域を割り付けるための命令(INT)
6. 制御の無条件または条件移行のためのジャンプ命令 (JMP, JPC). この命令は，if および while ステートメントで使われる.
7. 算術および関係演算命令の集合(OPR)

命令の形式は3つの成分，すなわち，命令コード f および2つの部分からなるパラメータ，に対する必要性から決められている(**図 5.8** 参照). 演算命令の場合には，a が演算内容を定めている；他の場合には，それは数 (LIT, INT), プログラム番地(JMP, JPC, CAL)あるいはデータ番地 (LOD, STO) である.

f	ℓ	a

図5.8 命令形式

PL/0 計算機の命令の詳細は，**プログラム 5.6** の一部である *interpret* とよばれる手続きから明らかであろう. プログラム 5.6 は完成されたコンパイラと解釈プログラムを1つのシステムに結合したものであり，PL/0 プログラムを翻訳し，実行する. このプログラムを修正して，既存の計算機に対するコードを生成することは，興味ある読者への練習問題としておく. その結果のコンパイラ・プログラムがどの程度の大きさになるかということは，選ばれたその計算機がこの仕事に対してどれ位適切であるかを計るめやすとなるであろう.

ここで述べた PL/0 計算機をさらに複雑な構造に拡張することにより，いくつかの演算をもっと能率良くすることが可能であることは疑う余地がない. その一例は番地指定機構である. ここで述べた方法が選ばれた理由は，その特有な簡単さと，どのような改良も本質的にはこれに基づき，またこれから導かれるということである.

5.11 コード生成

命令を組み立てることができるためには，コンパイラはその命令コードやパラメータを知っていなければならない. これらは整数リテラルあるいは番地である. これらの値は，コンパイラ自身によって，対応する識別名と関連づけられるが，この関連づけは定数，変数および手続き宣言を処理する際に行なわれ

る．この目的のために，識別名を記録している表を拡張して，各識別名に関連した属性を含ませるようにする．もし識別名が定数を表わす場合には，その属性値はその定数値である；もし識別名が変数ならば，その属性は番地であり，変位とレベルからなる；さらにもし識別名が手続きを表わすならば，その属性はその手続きの入口番地とレベルである．変数 *table* のこれに対応する拡張は，プログラム5.6の中に示されている．これは，データ宣言の段階的詳細化（または機能増強）がステートメント部の詳細化と同時に進行する顕著な例である．

定数の値はプログラム・テキストにより与えられるが，番地を決定するのはコンパイラの仕事である．PL/0 は十分に簡単な言語であるので，変数やコードの割付けは順次的に行なうことができる．したがって，すべての変数宣言はデータ割付けインデックスを1だけ増加させることにより処理される．（なぜなら PL/0 計算機の定義より，どの変数もちょうど1個の記憶セルを占有するからである）．データ割付けインデックス *dx* は，手続きのコンパイルを開始するときに初期化されるが，これはそのデータ領域が空から始まることを示している．[実際には *dx* には初期値3が与えられる．これは各データ領域が最低3個の変数 *RA, DL, SL*（前節参照）を含んでいるからである]．識別名の属性を決定するために必要な計算は，表に新しい識別名を加えるための手続き *enter* の中に含まれている．

オペランドについてのこれらの情報があれば，実際のコードを生成することは比較的簡単な仕事である．PL/0 計算機の便利なスタック機構のおかげで，ソース言語中のオペランドおよび演算子と目的コード中の命令の間には，実際上，1対1の対応がある．したがって，コンパイラは適当な再順序づけを行ない，**後置形式**に直すだけでよい．ここで“後置形式”とは，通常のように演算子をオペランドの中間に置く**中置形式**の代わりに，オペランドの後に演算子を置くことを意味する．後置形式は，（その発明者 Lukasciewicz にちなんで）**ポーランド形式**，または，括弧が不必要であることから，**括弧不要形式**と呼ばれることがある．式の中置形式と後置形式のいくつかの対応が**表5.4**に示されている（4.4.2節も参照せよ）．

この変換を行なうための非常に簡単な方法が，プログラム5.6中の手続き *expression* と *term* によって示されている．それは単に算術演算子の転送を遅らせることにすぎない．この点に関して，そこで示されている構文解析手続き群の配置は，種々の演算子間の通常の優先順位を適切に解釈したものになっていることに読者は注意してほしい．

これよりもほんのわずかだけ自明でない問題に，条件および繰返し文の翻訳

表 5.4 中置および後置形式の式

中置形式	後置形式
$x+y$ $(x-y)+z$ $x-(y+z)$ $x*(y+z)*w$	$xy+$ $xy-z+$ $xyz+-$ $xyz+*w*$

がある．この場合，ジャンプ命令の生成が必要であるが，時には行先番地が未知である場合がある．命令を厳密に順次的に出力ファイルに生成しようとするならば，**2パス** (two-pass) コンパイラ方式が必要になる．このとき，2回目のパスでは，不完全なジャンプ命令にその行先番地を補足する仕事を行なう．このコンパイラで採用されているもう1つの解決方法は，命令を配列の中に置き，本質的には直接アクセス可能な記憶領域にそれらを保持することである．この方法によれば，不明の番地がわかった時点で直ちにそれを補足することができる．この操作は，通常，修正(fixup)とよばれる．

そのような前向きのジャンプ命令を生成する際に行なうべき付加的操作は，その位置すなわちプログラム記憶部のインデックスを記録しておくことだけである．この番地は修正を行なう時に，不完全命令を捜すために利用される．その詳細は，再びプログラム 5.6 から明らかである．(*if* と *while* ステートメントを処理するルーチンを参照)．*if* と *while* ステートメントに対して生成されるコードのパターンは，次のようである ($L1$ と $L2$ はコードの番地を表わす)；

if C **then** S	**while** C **do** S
条件 C に対するコード	L1 : Cに対するコード
JPC L1	JPC L2
ステートメント S に対するコード	S に対するコード
L1: ...	JMP L1
	L2: ...

便宜上 *gen* とよばれる補助手続きが導入されている．その目的は，3つのパラメータに従って命令を組み立て，それを出力することである．それはコード・インデックス *cx* を自動的に増加させるが，この *cx* は次の命令を出すべき位置を表わしている．

一例として，乗算ルーチン(5.14)をコンパイルして得られたコードを記号形式で下に示す．右側のコメントは，単に説明のためのものである．

2	INT	0, 5	リンクおよび局所変数のために領域を割り付ける.
3	LOD	1, 3	*x*
4	STO	0, 3	*a*
5	LOD	1, 4	*y*
6	STO	0, 4	*b*
7	LIT	0, 0	0
8	STO	1, 5	*z*
9	LOD	0, 4	*b*
10	LIT	0, 0	0
11	OPR	0, 12	>
12	JPC	0, 29	
13	LOD	0, 4	*b*
14	OPR	0, 7	奇数
15	JPC	0, 20	
16	LOD	1, 5	*z*
17	LOD	0, 3	*a*
18	OPR	0, 2	+
19	STO	1, 5	*z*
20	LIT	0, 2	2
21	LOD	0, 3	*a*
22	OPR	0, 4	*
23	STO	0, 3	*a*
24	LOD	0, 4	*b*
25	LIT	0, 2	2
26	OPR	0, 5	/
27	STO	0, 4	*b*
28	JMP	0, 9	
29	OPR	0, 0	復帰

PL/0 手続き 5.14 に対するコード

一般のプログラム言語をコンパイルする際の作業の多くは，ここで述べたPL/0 計算機のための PL/0 コンパイラの場合より遙かに複雑である [5-4]. それらの多くは簡潔な組織化が非常に行ないにくいものである．ここで述べたコンパイラをより強力な言語に，または，より普通の計算機に拡張しようとするならば，このことの正しさがすぐに分るであろう．それにもかかわらず，複雑なプログラムを設計するためのここで述べた方法論は，その有効性を失うことはなく，むしろ，その作業が複雑かつ込み入ったものになればなる程，その価値を高めるであろう．実際，それは大きなコンパイラの作成に使われ，成功

をおさめている [5-1 および 5-9].

プログラム 5.6 PL/0 コンパイラ

```
program PL0(input,output);
{PL/0 compiler with code generation}
label 99;
const norw = 11;          {no. of reserved words}
   txmax = 100;           {length of identifier table}
   nmax = 14;             {max. no. of digits in numbers}
   al = 10;               {length of identifiers}
   amax = 2047;           {maximum address}
   levmax = 3;            {maximum depth of block nesting}
   cxmax = 200;           {size of code array}
type symbol =
   (nul, ident, number, plus, minus, times, slash, oddsym,
    eql, neq, lss, leq, gtr, geq, lparen, rparen, comma, semicolon,
    period, becomes, beginsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, constsym, varsym, procsym);
   alfa = packed array [1 . . al] of char;
   object = (constant, variable, procedure);
   symset = set of symbol;
   fct = (lit, opr, lod, sto, cal, int, jmp, jpc);     {functions}
   instruction = packed record
                    f: fct;            {function code}
                    l: 0 . . levmax;   {level}
                    a: 0 . . amax;     {displacement address}
                 end ;
{   LIT 0,a  :  load constant a
    OPR 0,a  :  execute operation a
    LOD l,a  :  load variable l,a
    STO l,a  :  store variable l,a
    CAL l,a  :  call procedure a at level l
    INT 0,a  :  increment t-register by a
    JMP 0,a  :  jump to a
    JPC 0,a  :  jump conditional to a    }
var ch: char;             {last character read}
   sym: symbol;           {last symbol read}
   id: alfa;              {last identifier read}
   num: integer;          {last number read}
```

```
    cc: integer;            {character count}
    ll: integer;            {line length}
    kk: integer;
    cx: integer;            {code allocation index}
    line: array [1 .. 81] of char;
    a: alfa;
    code: array [0 .. cxmax] of instruction;
    word: array [1 .. norw] of alfa;
    wsym: array [1 .. norw] of symbol;
    ssym: array [char] of symbol;
    mnemonic: array [fct] of
                    packed array [1 .. 5] of char;
    declbegsys, statbegsys, facbegsys: symset;
    table: array [0 .. txmax] of
            record name: alfa;
                case kind: object of
                constant: (val: integer);
                variable, procedure: (level, adr: integer)
            end ;
procedure error(n: integer);
begin writeln(' ****', ' ': cc−1, '↑', n: 2); err := err+1
end {error} ;

procedure getsym;
    var i,j,k: integer;

    procedure getch;
    begin if cc = ll then
        begin if eof(input) then
                begin write(' PROGRAM INCOMPLETE'); goto 99
                end ;
            ll := 0; cc := 0; write(cx: 5, ' ');
            while ¬eoln(input) do
                begin ll := ll+1; read(ch); write(ch); line[ll] := ch
                end ;
            writeln; ll := ll+1; read(line[ll])
        end ;
        cc := cc+1; ch := line[cc]
    end {getch} ;
```

```
begin {getsym}
   while ch = ' ' do getch;
   if ch in ['A' .. 'Z'] then
   begin {identifier or reserved word}  k := 0;
      repeat if k < al then
         begin k := k+1; a[k] := ch
         end ;
         getch
      until ¬(ch in ['A' .. 'Z','0' .. '9']);
      if k ≥ kk then kk := k else
         repeat a[kk] := ' '; kk := kk−1
         until kk = k;
      id := a; i := 1; j := norw;
      repeat k := (i+j) div 2;
         if id ≤ word[k] then j := k−1;
         if id ≥ word[k] then i := k+1
      until i > j:
      if i−1 > j then sym := wsym[k] else sym := ident
   end else
   if ch in ['0' .. '9'] then
   begin {number} k := 0; num := 0; sym := number;
      repeat num := 10*num + (ord(ch)−ord('0'));
         k := k+1; getch
      until ¬(ch in ['0' .. '9']);
      if k > nmax then error (30)
   end else
   if ch = ':' then
   begin getch;
      if ch = '=' then
      begin sym := becomes; getch
      end else sym := nul;
   end else
   begin sym := ssym[ch]; getch
   end
end {getsym} ;
```

```
procedure gen(x: fct; y,z: integer);
begin if cx > cxmax then
          begin write(' PROGRAM TOO LONG'); goto 99
          end ;
   with code[cx] do
      begin f := x; l := y; a := z
      end ;
   cx := cx + 1
end {gen} ;

procedure test(s1,s2: symset; n: integer);
begin if ¬(sym in s1) then
      begin error(n); s1 := s1 + s2;
         while ¬(sym in s1) do getsym
      end
end {test} ;

procedure block(lev,tx: integer; fsys: symset);
   var dx: integer;       {data allocation index}
      tx0: integer;       {initial table index}
      cx0: integer;       {initial code index}
   procedure enter(k: object);
   begin {enter object into table}
      tx := tx + 1;
      with table[tx] do
      begin name := id; kind := k;
         case k of
         constant: begin if num > amax then
                             begin error (31); num := 0 end ;
                      val := num
                   end ;
         variable: begin level := lev; adr := dx; dx := dx+1;
                   end ;
         procedure: level := lev
         end
      end
   end {enter} ;
```

```
    function position(id: alfa): integer;
        var i: integer;
    begin {find indentifier id in table}
        table[0].name := id; i := tx;
        while table[i].name ≠ id do i := i-1;
        position := i
    end {position};

    procedure constdeclaration;
    begin if sym = ident then
        begin getsym;
            if sym in [eql ,becomes] then
            begin if sym = becomes then error(1);
                getsym;
                if sym = number then
                    begin enter(constant); getsym
                    end
                else error (2)
            end else error (3)
        end else error (4)
    end {constdeclaration};

procedure vardeclaration;
begin if sym = ident then
        begin enter(variable); getsym
        end else error (4)
end {vardeclaration};

procedure listcode;
    var i: integer;
begin {list code generated for this block}
    for i := cx0 to cx-1 do
        with code[i] do
            writeln(i, mnemonic[f]:5, l:3, a:5)
end {listcode};

procedure statement(fsys: symset);
    var i,cx1,cx2: integer;
    procedure expression(fsys: symset);
        var addop: symbol;
        procedure term(fsys: symset);
            var mulop: symbol;
```

```
        procedure factor(fsys: symset);
          var i: integer;
        begin test(facbegsys, fsys, 24);
          while sym in facbegsys do
          begin
            if sym = ident then
            begin i := position(id);
              if i = 0 then error (11) else
              with table[i] do
              case kind of
                constant: gen(lit, 0, val);
                variable: gen(lod, lev-level, adr);
                procedure: error (21)
              end ;
              getsym
            end else
            if sym = number then
            begin if num > amax then
                  begin error (31); num := 0
                  end ;
              gen(lit, 0, num); getsym
            end else
            if sym = lparen then
            begin getsym; expression([rparen]+fsys);
              if sym = rparen then getsym else error (22)
            end ;
            test(fsys, [lparen], 23)
          end
        end {factor} ;
    begin {term} factor(fsys+[times, slash]);
      while sym in [times, slash] do
        begin mulop := sym; getsym; factor(fsys+[times, slash]);
          if mulop = times then gen(opr,0,4) else gen(opr,0,5)
        end
    end {term} ;
begin {expression}
    if sym in [plus, minus] then
      begin addop := sym; getsym; term(fsys+[plus, minus]);
        if addop = minus then gen(opr,0,1)
      end else term(fsys+[plus, minus]);
```

```
        while sym in [plus, minus] do
            begin addop := sym; getsym; term(fsys+[plus, minus]);
                if addop = plus then gen(opr,0,2) else gen(opr,0,3)
            end
    end {expression} ;

    procedure condition(fsys: symset);
        var relop: symbol;
    begin
        if sym = oddsym then
        begin getsym; expression(fsys); gen(opr,0,6)
        end else
        begin expression([eql, neq, lss, gtr, leq, geq]+fsys);
            if ¬(sym in [eql, neq, lss, leq, gtr, geq]) then
                error (20) else
            begin relop := sym; getsym; expression(fsys);
                case relop of
                  eql: gen(opr,0, 8);
                  neq: gen(opr,0, 9);
                  lss: gen(opr,0,10);
                  geq: gen(opr,0,11);
                  gtr: gen(opr,0,12);
                  leq: gen(opr,0,13);
                end
            end
        end
    end {condition} ;
begin {statement}
    if sym = ident then
    begin i := position(id);
        if i = 0 then error (11) else
        if table[i] .kind ≠ variable then
            begin {assignment to non-variable} error (12); i := 0
            end ;
        getsym; if sym = becomes then getsym else error (13);
        expression(fsys);
        if i ≠ 0 then
            with table[i] do gen(sto, lev-level, adr)
    end else
```

```
    if sym = callsym then
    begin getsym;
        if sym ≠ ident then error (14) else
            begin i := position(id);
                if i = 0 then error (11) else
                with table[i] do
                    if kind = procedure then gen (cal, lev-level, adr)
                    else error (15);
                getsym
            end
    end else
    if sym = ifsym then
    begin getsym; condition([thensym, dosym]+fsys);
        if sym = thensym then getsym else error (16);
        cx1 := cx; gen(jpc,0,0);
        statement(fsys); code[cx1].a := cx
    end else
    if sym = beginsym then
    begin getsym; statement([semicolon, endsym]+fsys);
        while sym in [semicolon]+statbegsys do
        begin
            if sym = semicolon then getsym else error (10);
            statement([semicolon, endsym]+fsys)
        end ;
        if sym = endsym then getsym else error (17)
    end else
    if sym = whilesym then
    begin cx1 := cx; getsym; condition([dosym]+fsys);
        cx2 := cx; gen(jpc,0,0);
        if sym = dosym then getsym else error (18);
        statement(fsys); gen(jmp,0,cx1); code[cx2].a := cx
    end ;
    test(fsys, [ ], 19)
end {statement} ;
begin {block} dx := 3; tx0 := tx; table[tx] .adr := cx; gen(jmp,0,0);
    if lev > levmax then error (32);
    repeat
        if sym = constsym then
```

```
        begin getsym;
            repeat constdeclaration;
                while sym = comma do
                    begin getsym; constdeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident
        end ;
        if sym = varsym then
        begin getsym;
            repeat vardeclaration;
                while sym = comma do
                    begin getsym; vardeclaration
                    end ;
                if sym = semicolon then getsym else error (5)
            until sym ≠ ident;
        end ;
        while sym = procsym do
        begin getsym;
            if sym = ident then
                begin enter(procedure); getsym
                end
            else error (4);
            if sym = semicolon then getsym else error (5);
            block(lev+1,tx,[semicolon]+fsys);
            if sym = semicolon then
                begin getsym; test(statbegsys+[ident, procsym], fsys, 6)
                end
            else error (5)
        end ;
        test(statbegsys+[ident], declbegsys, 7)
    until ¬(sym in declbegsys);
    code[table[tx0].adr].a := cx;
    with table[tx0] do
        begin adr := cx; {start adr of code}
        end ;
    cx0 := cx; gen(int,0,dx);
    statement([semicolon, endsym]+fsys);
    gen(opr,0,0); {return}
    test(fsys, [ ], 8);
    listcode;
end {block} ;
```

```
procedure interpret;
   const stacksize = 500;
   var p,b,t: integer {program-, base-, topstack-registers}
      i: instruction; {instruction register}
      s: array [1 . . stacksize] of integer;  {datastore}
   function base(l: integer): integer;
      var b1: integer;
   begin b1 := b; {find base l levels down}
      while l > 0 do
         begin b1 := s[b1]; l := l-1
         end ;
      base := b1
   end {base} ;

begin writeln(' START PL/0');
   t := 0; b := 1; p := 0;
   s[1] := 0; s[2] := 0; s[3] := 0;
   repeat i := code[p]; p := p+1;
      with i do
      case f of
     lit: begin t := t+1; s[t] := a
          end ;
     opr: case a of      {operator}
           0: begin {return}
                 t := b-1; p := s[t+3]; b := s[t+2];
              end ;
           1: s[t] := -s[t];
           2: begin t := t-1; s[t] := s[t] + s[t+1]
              end ;
           3: begin t := t-1; s[t] := s[t] - s[t+1]
              end ;
           4: begin t := t-1; s[t] := s[t] * s[t+1]
              end ;
           5: begin t := t-1; s[t] := s[t] div s[t+1]
              end ;
           6: s[t] := ord(odd(s[t]));
           8: begin t := t-1; s[t] := ord(s[t]=s[t+1])
              end ;
```

```
            9: begin t := t-1; s[t] := ord(s[t]≠s[t+1])
               end ;
           10: begin t := t-1; s[t] := ord(s[t]<s[t+1])
               end ;
           11: begin t := t-1; s[t] := ord(s[t]≥s[t+1])
               end ;
           12: begin t := t-1; s[t] := ord(s[t]>s[t+1])
               end ;
           13: begin t := t-1; s[t] := ord(s[t]≤s[t+1])
               end ;
           end ;
     lod: begin t := t+1; s[t] := s[base(l)+a]
          end ;
     sto: begin s[base(l)+a] := s[t]; writeln(s[t]); t := t-1
          end ;
     cal: begin {generate new block mark}
               s[t+1] := base(l); s[t+2] := b; s[t+3] := p;
               b := t+1; p := a
           end ;
      int: t := t+a;
     jmp: p := a;
      jpc: begin if s[t] = 0 then p := a; t := t-1
           end
       end {with, case}
   until p = 0;
   write(' END PL/0');
end {interpret} ;

begin {main program}
   for ch := 'A' to ';' do ssym[ch] := nul;
   word[ 1] := 'BEGIN ';   word[ 2] := 'CALL      ';
   word[ 3] := 'CONST';    word[ 4] := 'DO        ';
   word[ 5] := 'END   ';   word[ 6] := 'IF        ';
   word[ 7] := 'ODD   ';   word[ 8] := 'PROCEDURE';
   word[ 9] := 'THEN  ';   word[10] := 'VAR       ';
   word[11] := 'WHILE ';
   wsym[ 1] := beginsym; wsym[ 2] := callsym;
   wsym[ 3] := constsym; wsym[ 4] := dosym;
   wsym[ 5] := endsym;   wsym[ 6] := ifsym;
   wsym[ 7] := oddsym;   wsym[ 8] := procsym;
```

```
   wsym[ 9] := thensym;  wsym[10] := varsym;
   wsym[11] := whilesym;
   ssym['+'] := plus;       ssym['-'] := minus;
   ssym['*']  := times;     ssym['/']  := slash;
   ssym['(']  := lparen;    ssym[')']  := rparen;
   ssym['='] := eql;        ssym[',']  := comma;
   ssym['.']  := period;    ssym['≠'] := neq;
   ssym['<'] := lss;        ssym['>'] := gtr;
   ssym['≤'] := leq;        ssym['≥'] := geq;
   ssym[';']  := semicolon;
   mnemonic[lit]  := 'LIT '; mnemonic[opr] := 'OPR';
   mnemonic[lod]  := 'LOD'; mnemonic[sto]  := 'STO';
   mnemonic[cal]  := 'CAL'; mnemonic[int]  := 'INT ';
   mnemonic[jmp] := 'JMP'; mnemonic[jpc]  := 'JPC';
   declbegsys := [constsym, varsym, procsym];
   statbegsys := [beginsym, callsym, ifsym, whilesym];
   facbegsys := [ident, number, lparen];
   page(output);
   cc := 0; cx := 0; ll := 0; ch := ' '; kk := al; getsym;
   block(0,0,[period]+declbegsys+statbegsys);
   if sym ≠ period then error (9);
   if err = 0 then interpret else write(' ERRORS IN PL/0 PROGRAM');
99: writeln
end
```

演 習 問 題

5.1　次の構文則を考える.

$$S ::= A$$
$$A ::= B \mid \textbf{if } A \textbf{ then } A \textbf{ else } A$$
$$B ::= C \mid B+C \mid +C$$
$$C ::= D \mid C*D \mid *D$$
$$D ::= x \mid (A) \mid -D$$

終端記号と非終端記号はどれか. 各非終端記号 X に対して, 最左の記号および後続記号の集合 $L(X)$ と $F(X)$ を決定せよ. 次の文に対する構文解

析ステップの列を構成せよ.

$$x+x$$
$$(x+x)*(+-x)$$
$$(x*-+x)$$
$$\textbf{if } x+x \textbf{ then } x*x \textbf{ else} -x$$
$$\textbf{if } x \textbf{ then if} -x \textbf{ then } x \textbf{ else } x+x \textbf{ else } x*x$$
$$\textbf{if} -x \textbf{ then } x \textbf{ else if } x \textbf{ then } x+x \textbf{ else } x$$

5.2 演習問題5.1の文法は，1記号の先読みによるトップダウン構文解析のための制約規則1と2を満たすか．もしそうでないならば，これらの規則を満たす等価な構文則を見つけよ．構文グラフおよびプログラム5.3で使ったデータ構造によって，この構文則を表現せよ．

5.3 次の構文則に対して演習問題5.2を再度行なえ．

$$S ::= A$$
$$A ::= B \,|\, \textbf{if } C \textbf{ then } A \,|\, \textbf{if } C \textbf{ then } A \textbf{ else } A$$
$$B ::= D = C$$
$$C ::= \textbf{if } C \textbf{ then } C \textbf{ else } C \,|\, D$$

ヒント：1記号の先読みによるトップダウン構文解析が適用できるためには，いくつかの構文を削除したり取り換える必要のあることがわかるであろう．

5.4 次の構文則が与えられたとき，トップダウン構文解析の問題を考える．

$$S ::= A$$
$$\mathrm{A} ::= B+A \,|\, DC$$
$$B ::= D \,|\, D*B$$
$$D ::= x \,|\, (C)$$
$$C ::= +x \,|\, -x$$

この構文則に従って文を解析するためには，最大何個の記号を先読みしなければならないか．

5.5 PL/0の記述(図5.4)を等価なBNF生成規則に変換せよ．

5.6 与えられた生成規則の集合に対して，各非終端記号 S の先頭および後続記号集合 $L(S)$ と $F(S)$ を決定するプログラムを書け．

ヒント：プログラム 5.3 の一部分を用いて，構文則の内部表現をデータ構造の形に構成せよ．次に，このリンクされたデータ構造を操作せよ．

5.7　PL/0 とそのコンパイラに次の機能を加えよ：

(a)　次の形の条件文

<*statement*> : : = **if** <*condition*> **then** <*statement*> **else** <*statement*>

(b)　次の形の繰返し文

<*statement*> : : = **repeat** <*statement*> {;<*statement*>} **until** <*condition*>

与えられた PL/0 機能の形式や解釈に変更をもたらすような特別な問題点があるか．PL/0 計算機に新しい命令を追加してはいけない．

5.8　PL/0 言語とコンパイラを拡張して，手続きにパラメータを導入せよ．可能な 2 つの解を考え，そのうちのいずれかを選んで実現せよ．

(a)　**値パラメータ**　呼び出し中の実パラメータは式であり，その値は，手続きの頭部において指定される仮パラメータの表わす局所変数に代入される．

(b)　**変数パラメータ**　実パラメータは変数である．呼び出しの際，仮パラメータはそれで置き換えられる．変数パラメータは，実パラメータの番地を渡すことにより実現されるが，この番地は，仮パラメータによって示される場所に格納される．そして，この渡された番地を介して，実パラメータは間接的にアクセスされる．したがって，変数パラメータは，その手続きの外側で定義された変数へのアクセスを可能にする．ゆえに，変数の有効範囲に関する規則は次のように変更してもよい：どの手続きにおいても，局所変数だけが直接アクセス可能であり，非局所変数はパラメータを介してのみアクセス可能である．

5.9　PL/0 言語とコンパイラを拡張して配列変数を導入せよ．配列変数 a のインデックスの範囲は，その宣言において次のように示されるとする．

var a(*low* : *high*)

5.10　PL/0 コンパイラを修正し，諸君の使える計算機用のコードを生成するようにせよ．

ヒント　ローダに関する規約の問題を避けるために，記号アセンブラ・コードを生成せよ．最初の段階では，たとえばレジスタの使用法に関して，コードを最適化することは避けよ．最適化は，コンパイラの詳細化の第 4

番目の段階において組み込むべきである.

5.11 プログラム 5.5 を "prettyprint" とよばれるプログラムに拡張せよ. このプログラムの目的は, PL/0 テキストを読み込み, 行の区切りとインデンテーション (行の頭を下げること) を適当に行なって, テキストの構造を自然に反映した形式でそれらを印刷することである. まず初めに PL/0 の構文構造にもとづいて, 行の区切りとインデンテーションのための正確な規則を定義せよ. 次に, プログラム 5.5 の上に印刷ステートメントを加えることにより, それらを実現せよ (もちろん走査プログラムからは印刷ステートメントを取り除かなければならない).

参考文献

5-1. AMMANN, U., "The Method of Structured Programming Applied to the Development of a Compiler," *International Computing Symposium 1973*, A. Günther et al. eds., (Amsterdam: North-Holland Publishing Co., 1974), pp. 93–99.

5-2. COHEN, D. J. and GOTLIEB, C. C., "A List Structure Form of Grammars for Syntactic Analysis," *Comp. Surveys*, **2**, No. 1 (1970), 65–82.

5-3. FLOYD, R. W., "The Syntax of Programming Languages—A Survey," *IEEE Trans.*, EC-13 (1964), 346–53.

5-4. GRIES, D., *Compiler Construction for Digital Computers* (New York: Wiley, 1971).

5-5. KNUTH, D. E., "Top-down Syntax Analysis," *Acta Informatica*, **1**, No. 2 (1971), 79–110.

5-6. LEWIS, P. M. and STEARNS, R. E., "Syntax-directed Transduction," *J. ACM*, **15**, No. 3 (1968), 465–88.

5-7. NAUR, P., ed., "Report on the Algorithmic Language ALGOL 60," *ACM*, **6**, No. 1 (1963), 1–17.

5-8. SCHORRE, D. V., "META II, A Syntax-oriented Compiler Writing Language," *Proc. ACM Natl. Conf.*, **19**, (1964), D 1.3.1-11.

5-9. WIRTH, N., "The Design of a PASCAL Compiler," *Software-Practice and Experience*, **1**, No. 4 (1971), 309–33.

補　遺　I

ASCII 文字集合

y \ x	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	;	K	[	k	{
12	ff	fs	,	<	L	\	l	|
13	cr	gs	−	=	M	]	m	}
14	so	rs	.	>	N	↑	n	~
15	si	us	/	?	O	_	o	del

文字 *ch* の順序数は，表中のその座標から次のように計算される．

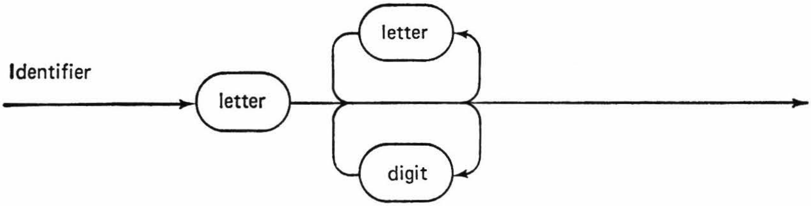
$$ord(ch) = 16*x + y$$

順序数が 0 から 31 および 127 の文字は，いわゆる **制御文字** であり，データ伝送や装置制御に使われる．順序数が 32 の文字は空白文字である．

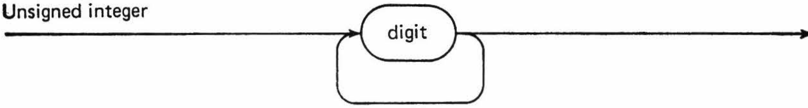
補　遺　II

PASCAL 構文グラフ

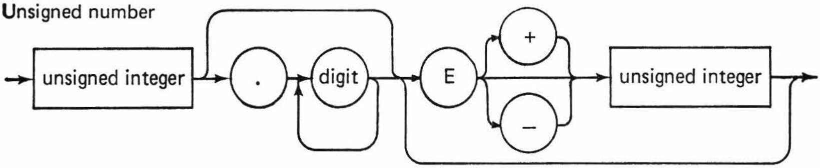
Identifier



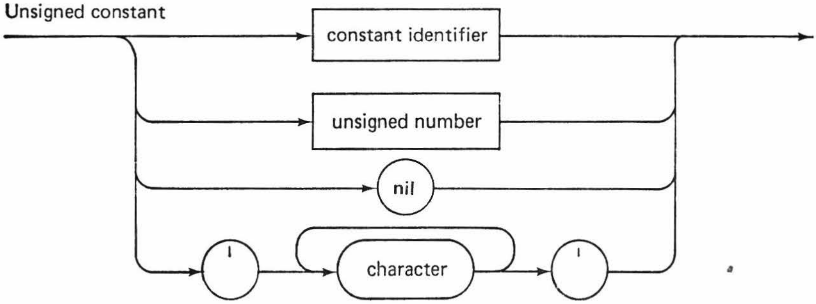
Unsigned integer



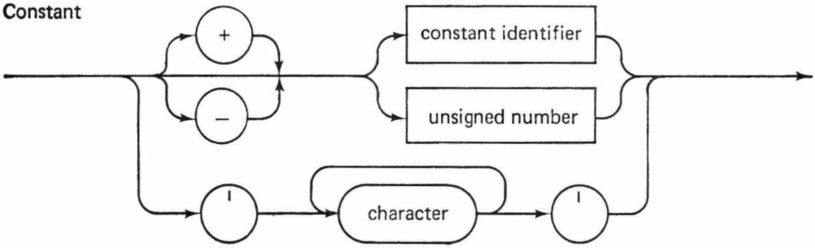
Unsigned number



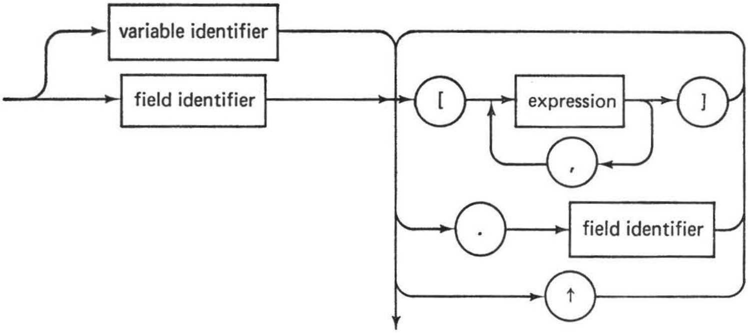
Unsigned constant



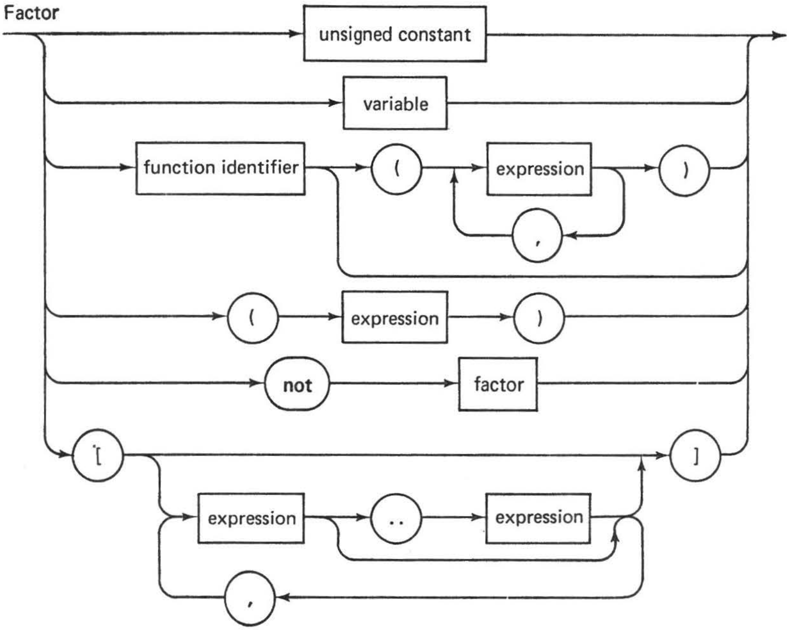
Constant



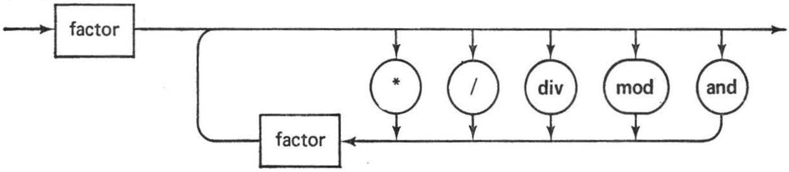
Variable



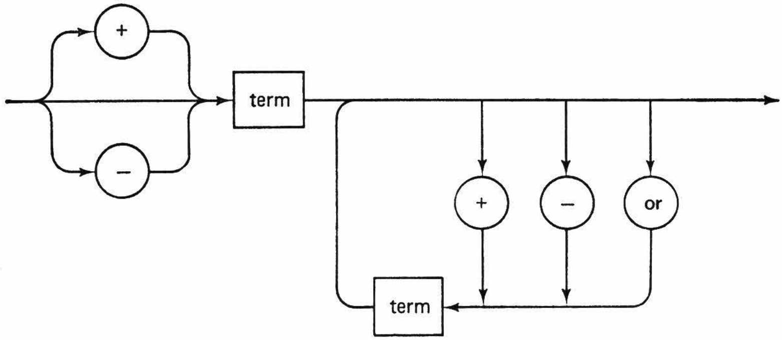
Factor



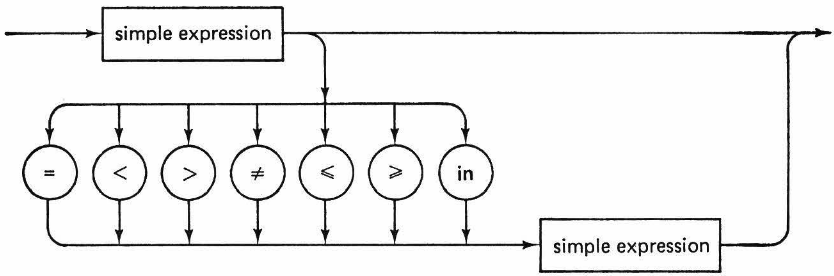
Term



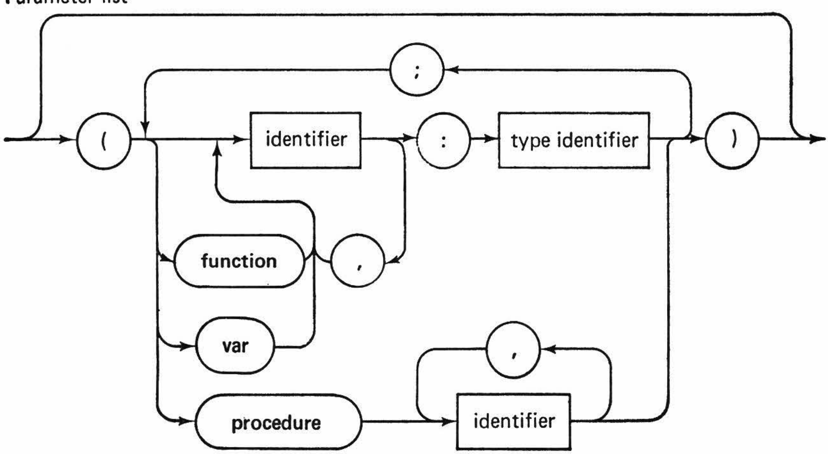
Simple expression



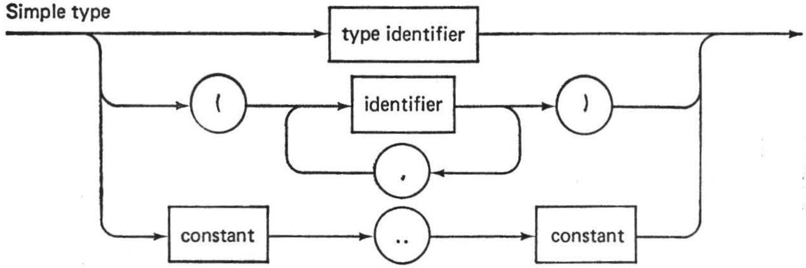
Expression



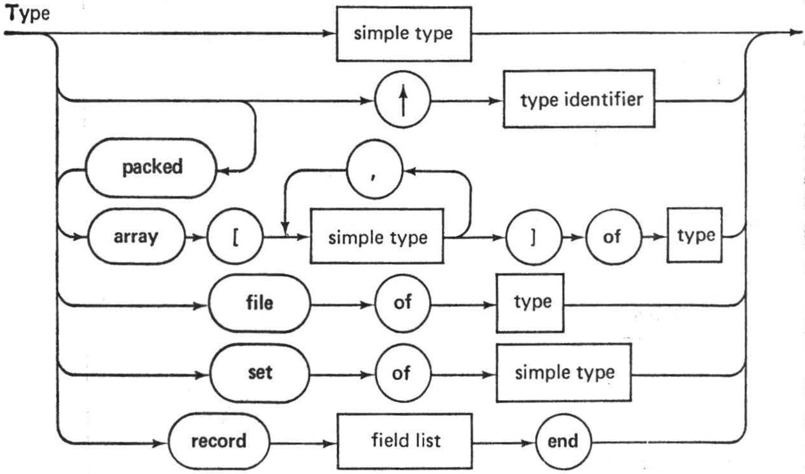
Parameter list



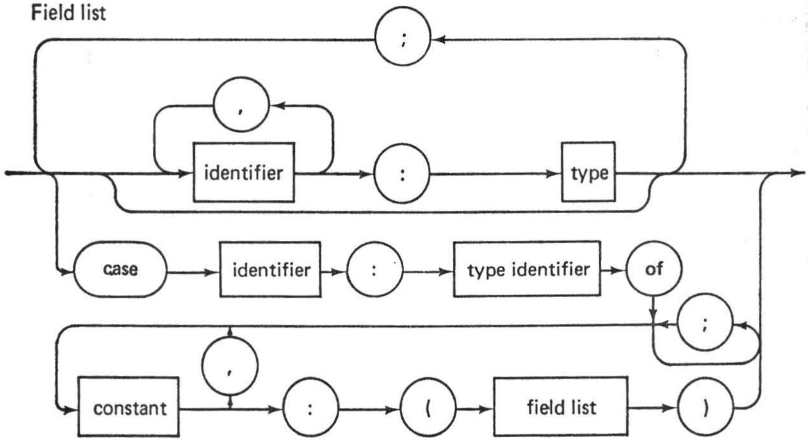
Simple type

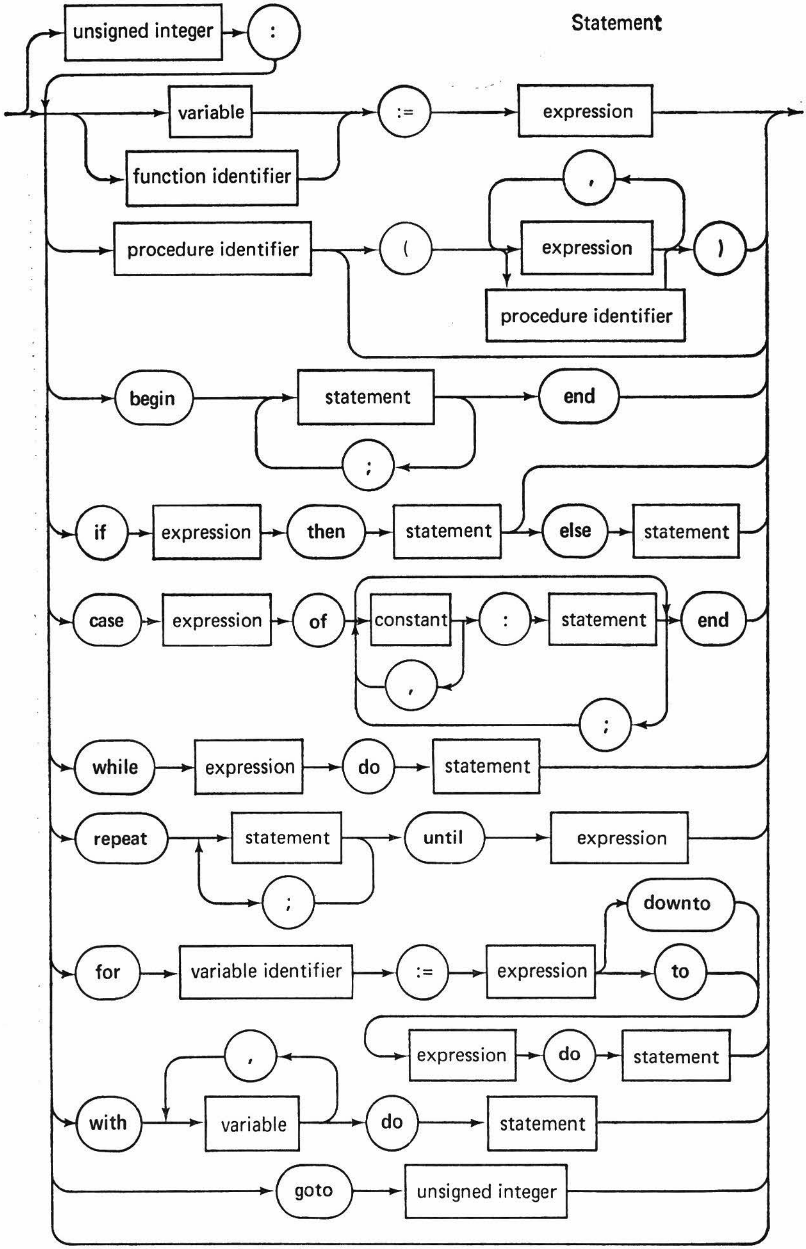


Type

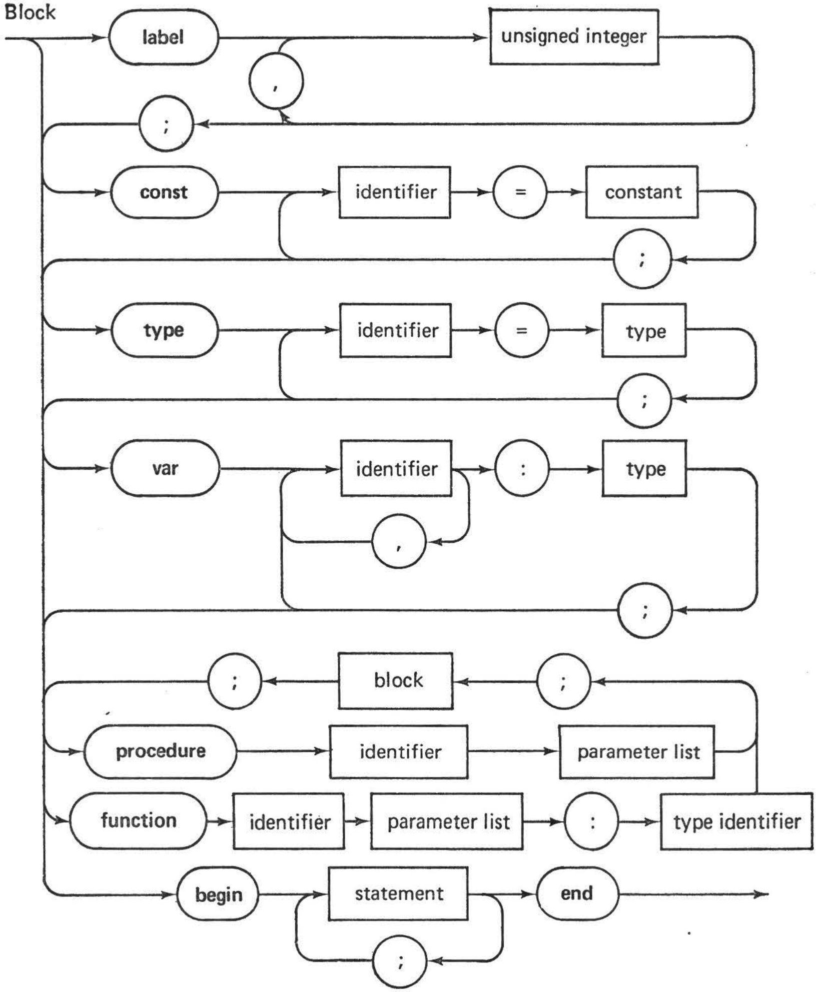


Field list

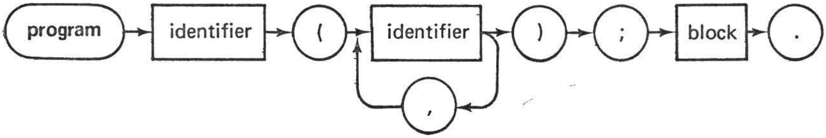



Statement
unsigned integer
:
variable
:=
expression
function identifier
procedure identifier
(
,
expression
)
procedure identifier
begin
statement
end
;
if
expression
then
statement
else
statement
case
expression
of
constant
:
statement
end
,
;
while
expression
do
statement
repeat
statement
until
expression
;
for
variable identifier
:=
expression
downto
to
expression
do
statement
with
,
variable
do
statement
goto
unsigned integer

Block



Program



索　　　引

英字索引およびプログラム索引は，用語索引の末尾に記載されている．

用語索引

あ　行

い

お

か　行

き

く

け

こ

英 字 索 引

プログラム索引

MEMO

MEMO

≪訳者略歴≫

片山卓也
かたやまたくや

1939年　東京都に生まれる.
1962年　東京工業大学理工学部電気工学科卒業
1964年　東京工業大学大学院理工学研究科修士課程修了,
日本アイ・ビー・エム株式会社入社
1967年　東京工業大学助手
1974年　東京工業大学助教授
現在:　**東京工業大学工学部情報工学科助教授**

アルゴリズム+データ構造=プログラム

昭和54年12月15日　初版発行　© 1979　科学技術出版社

訳者　片山　卓也
発行者　村越　勝弘

検	印
廃	止

発行所　**日本コンピュータ協会**
東京都文京区本郷3丁目5-5
〒113　電話(03)813-6929番

発売元　**科学技術出版社**
東京都文京区本郷3丁目5-5
〒113　電話(03)813-6920番

乱丁本・落丁本はお買求め書店にてお取りかえいたします.